
The Daily Design Pattern
Learning 22 Software Design Patterns in 22 Days
Matthew P Jones

Table of Contents

How To Use This Book

This book presents 22 of the Gang of Four's software design patterns, and walks you through creating an example of each of those patterns. The source code for this book can be found in the Daily Design Pattern GitHub repository; feel free to leave any comments there or on the corresponding blog posts.

If you wish to learn the patterns and do the examples, I recommend reading only one or two sections per day; otherwise, the patterns have a tendency to blend together, and some of the details about when and why they are used might become blurred.

All of the examples in this book are in C#, but the concepts shown by the design patterns themselves can be applied to any programming language or environment. One caveat: they were specifically designed for object-oriented programming languages, and so will be the most useful for those kinds of projects.

Who Are You Anyway?

My name is Matthew P Jones, and I'm a blogger at Exception Not Found. Among other things, I currently work as a full-time lead software developer for U-Haul International.

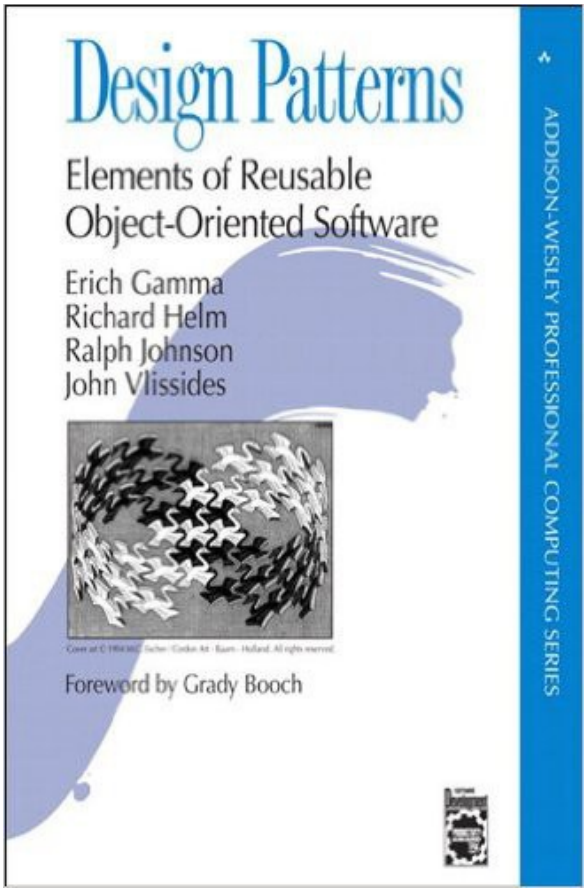
This series (The Daily Design Pattern) originated as a presentation I did for my coworkers, and grew to include a series of blog posts and, eventually, this very book you are reading now. My sincere hope is that this book, along with the blog posts and other presentation materials, will help foster a new understanding of software design patterns and how we use them in today's modern software development landscape.

What I'm Leaving Out

I won't be covering the Interpreter pattern, which is a bona fide Gang of Four pattern from their book, but its usage is so specific that it's been unusable for any project I've ever worked on. I don't like to teach what I don't understand myself, and because I don't understand how to use Interpreter, I won't be including it in this book. If you'd like to learn how to use this pattern, check out Do Factory.

Day 0: What Are Design Patterns?

Software design patterns are common solutions to problems which are regularly encountered in programming. These particular patterns deal with object-oriented programming exclusively, so applying these patterns to, say, a functional environment is a thoroughly bad idea. Some pattern proponents even go so far as to say that, in the object-oriented world, these design patterns are full-fledged best practices, though I often stop short of such an assertion.



The patterns I'll be describing in this series originate from a book titled, appropriately enough, Design Patterns - Elements of Reusable Object-Oriented Software, written by a group of authors who have come to be known as the Gang of Four (GoF). These authors are Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. The original book was released in 1995, but despite what you may think it didn't really create new patterns so much as give names to ones that already occurred naturally, so to speak.

That said, given that the examples in the book were given in C++ and Smalltalk, I thought I personally would be better served by having examples in modern C#, so that's what my examples will be written in.

As the Gang of Four note in their book, there are three types of design patterns:

- **Creational patterns** deal with the creation of objects and instances.
- **Structural patterns** deal with the structure of classes and code.
- **Behavioral patterns** deal with the behavior of objects.

So, without further ado, turn the page to begin your journey with Design Patterns by learning about one of the most common Creational patterns: Factory Method!

Day 1: Factory Method

What Is This Pattern?

The Factory Method design pattern is a Creational design pattern which defines an interface for creating an object, but doesn't specify what objects the individual implementations of that interface will instantiate.

All that means is that when using this pattern, you can define certain methods and properties of object that will be common to all objects created using the Factory Method, but let the individual Factory Methods define what specific objects they will instantiate.

The Rundown

- **Type:** Creational
- **Useful?** 5/5 (Extremely)
- **Good For:** Creating objects in a related family.
- **Example Code:** On GitHub
- **Blog Post:** "Factory Method - The Daily Design Pattern"

The Participants

- The **Product** defines the interfaces of objects that the factory method will create.
- The **ConcreteProduct** objects implement the Product interface.
- The **Creator** declares the factory method, which returns an object of type Product. The Creator can also define a default implementation of the factory method, though we will not see that in the below example.
- The **ConcreteCreator** objects overrides the factory method to return an instance of a Concrete Product.

A Delicious Example

To demo how this pattern works, let's talk about sandwiches.

According to Wikipedia, a sandwich...

> "...is a food item consisting of one or more types of food, such as vegetables, sliced cheese or meat, placed on or between slices of bread, or more generally any dish wherein two or more pieces of bread serve as a container or wrapper for some other food."

So, if we put two pieces of bread around anything edible, it becomes a sandwich. Yes, that

means hot dogs are sandwiches. I know, I was surprised too.

Let's build some classes to demo how we can use Factory Method to create a variety of different sandwiches. In this chapter, we'll say that Sandwiches are comprised of Ingredients. We'll need an abstract class Ingredient to represent this, and said Ingredient class does double-duty as our **Product** participant:

```
/// <summary>
/// Product
/// </summary>
abstract class Ingredient { }
```

Now let's instantiate a few classes to represent common ingredients in sandwiches (our **ConcreteProduct** participants):

```
/// <summary>
/// Concrete Product
/// </summary>
class Bread : Ingredient { }

/// <summary>
/// Concrete Product
/// </summary>
class Turkey : Ingredient { }

/// <summary>
/// Concrete Product
/// </summary>
class Lettuce : Ingredient { }

/// <summary>
/// Concrete Product
/// </summary>
class Mayonnaise : Ingredient { }
```

What we want to do is build a factory that will allow us to build different kinds of sandwiches using the same set of ingredients. What will differ between the kinds of sandwiches will be the amount and order of said ingredients.

First, let's build an abstract class `Sandwich` that represents all possible kinds of sandwiches (this is the **Creator** participant).

In the code snippet on the next page, note the `CreateIngredients()` method; this method is the Factory Method which gives the pattern its name. It's not implemented here because that implementation is left up to the ConcreteCreator classes that we now need to define.

```
/// <summary>
/// Creator
/// </summary>
abstract class Sandwich
```

```
{
    private List<Ingredient> _ingredients = new List<Ingredient>();

    public Sandwich()
    {
        CreateIngredients();
    }

    //Factory method
    public abstract void CreateIngredients();

    public List<Ingredient> Ingredients
    {
        get { return _ingredients; }
    }
}
```

Now that we've got our Creator participant defined, we can build some **ConcreteCreator** classes. Let's start off with a basic turkey sandwich:

```
/// <summary>
/// Concrete Creator
/// </summary>
class TurkeySandwich : Sandwich
{
    public override void CreateIngredients()
    {
        Ingredients.Add(new Bread());
        Ingredients.Add(new Mayonnaise());
        Ingredients.Add(new Lettuce());
        Ingredients.Add(new Turkey());
        Ingredients.Add(new Turkey());
        Ingredients.Add(new Bread());
    }
}
```

Whenever we create an object of class `TurkeySandwich`, we can call `CreateIngredients()` to create the correct amount and order of ingredients for this sandwich.

But what if we wanted to go... *bigger*? Like, say, instead of one layer in a sandwich, how about ten layers?

The image to the right shows what's known in the United States as a Dagwood sandwich, named after a comic strip character who was fond of making them. A Dagwood is a ridiculously large sandwich, with many layers of bread and fillings.

We want to create a class to represent a Dagwood sandwich. What makes the Factory Method design pattern so useful is that, in order to create a new class for a Dagwood, all we need to do is instantiate a class and override the `CreateIngredients()` method, like so:

```
/// <summary>
/// Concrete Creator
/// </summary>
class Dagwood : Sandwich //OM NOM NOM
{
    public override void CreateIngredients()
    {
        Ingredients.Add(new Bread());
        Ingredients.Add(new Turkey());
        Ingredients.Add(new Turkey());
        Ingredients.Add(new Lettuce());
        Ingredients.Add(new Lettuce());
        Ingredients.Add(new Mayonnaise());
        Ingredients.Add(new Bread());
        Ingredients.Add(new Turkey());
        Ingredients.Add(new Turkey());
        Ingredients.Add(new Lettuce());
        Ingredients.Add(new Lettuce());
        Ingredients.Add(new Mayonnaise());
        Ingredients.Add(new Bread());
        Ingredients.Add(new Turkey());
        Ingredients.Add(new Turkey());
        Ingredients.Add(new Lettuce());
        Ingredients.Add(new Lettuce());
        Ingredients.Add(new Mayonnaise());
        Ingredients.Add(new Bread());
        Ingredients.Add(new Turkey());
        Ingredients.Add(new Turkey());
        Ingredients.Add(new Lettuce());
        Ingredients.Add(new Lettuce());
        Ingredients.Add(new Mayonnaise());
        Ingredients.Add(new Bread());
        Ingredients.Add(new Turkey());
        Ingredients.Add(new Turkey());
        Ingredients.Add(new Lettuce());
        Ingredients.Add(new Lettuce());
        Ingredients.Add(new Mayonnaise());
        Ingredients.Add(new Bread());
        Ingredients.Add(new Turkey());
        Ingredients.Add(new Turkey());
        Ingredients.Add(new Lettuce());
        Ingredients.Add(new Lettuce());
        Ingredients.Add(new Mayonnaise());
        Ingredients.Add(new Bread());
    }
}
```

Now, when we want to create objects of type `TurkeySandwich` or `Dagwood`, we simply call them like this:

```
class Program
{
    static void Main(string[] args)
    {
        var turkeySandwich = new TurkeySandwich();
        var dagwood = new Dagwood();
        //Do something with these sandwiches (like, say, eat them).
        ...
    }
}
```

Will I Ever Use This Pattern?

Absolutely. The Factory Method pattern is exceedingly common in today's software design world. Any time you need to create groups of related objects, Factory Method is one of the cleanest ways to do so.

Summary

The Factory Method pattern provides a manner in which we can instantiate objects, but the details of the creation of those instance are left to be defined by the instance classes themselves. This pattern is best used when you need to create lots of objects which are in the same family.

Day 2: Abstract Factory

What Is This Pattern?

The Abstract Factory Pattern (AKA Factory of Factories) is a Creational pattern in which interfaces are defined for creating families of related objects without specifying their actual implementations.

When using this pattern, you create factories which return many kinds of related objects. This pattern enables larger architectures such as Dependency Injection.

The Rundown

- **Type:** Creational
- **Useful?** 5/5 (Absolutely)
- **Good For:** Creating objects in different related families without relying on concrete implementations.
- **Example Code:** On GitHub

- **Blog Post:** "Abstract Factory - The Daily Design Pattern"

The Participants

- The **AbstractFactory** declares an interface for operations which will create AbstractProduct objects.
- The **ConcreteFactory** objects implement the operations defined by the AbstractFactory.
- The **AbstractProduct** declares an interface for a type of product.
- The **Products** define a product object that will be created by the corresponding ConcreteFactory.
- The **Client** uses the AbstractFactory and AbstractProduct interfaces.

A Delicious Example

On Day 1, we modeled the Factory Method design pattern using sandwiches. The thing about sandwiches is that they no matter what they are made of (turkey, roast beef, veggies, peanut butter and jelly) they're still sandwiches, e.g. something edible between two slices of bread. In the Factory Method example, sandwiches could be considered a family of related objects.

But what if wanted to model *several* families of objects, not just one? To demo the Abstract Factory design pattern, let's go more general and model entire sets of recipes.

Let's say we want to model two kinds of recipes: a Sandwich and a Dessert. Further, let's make the assumption that adults and kids don't eat the same things, and so we want one of each kind of recipe for adults and children.

To demo this, let's make some abstract classes representing the generic kinds of recipes (these are our **AbstractProduct** participants):

```
/// <summary>
/// An abstract object.
/// </summary>
abstract class Sandwich { }

/// <summary>
/// An abstract object.
/// </summary>
abstract class Dessert { }
```

Next, we need an abstract class that will return a Sandwich and a Dessert (this is the **AbstractFactory** participant):

```
/// <summary>
/// The AbstractFactory class, which defines methods for creating abstract
/// objects.
/// </summary>
abstract class RecipeFactory
{
    public abstract Sandwich CreateSandwich();
    public abstract Dessert CreateDessert();
}
```

Now we can start implementing the actual objects. First let's consider the adult menu (these next classes are **ConcreteProduct** objects):

```
/// <summary>
/// A ConcreteProduct
/// </summary>
class BLT : Sandwich { }

/// <summary>
/// A ConcreteProduct
/// </summary>
class CremeBrulee : Dessert { }
```

We also need a **ConcreteFactory** which implements the AbstractFactory and returns the adult recipes:

```
/// <summary>
/// A ConcreteFactory which creates concrete objects by implementing the
/// abstract factory's methods.
/// </summary>
class AdultCuisineFactory : RecipeFactory
{
    public override Sandwich CreateSandwich()
    {
        return new BLT();
    }

    public override Dessert CreateDessert()
    {
        return new CremeBrulee();
    }
}
```

Now that we've got the Adult recipes defined, let's define the Child recipes. Here are the **ConcreteProduct** classes and **ConcreteFactory** for said recipes:

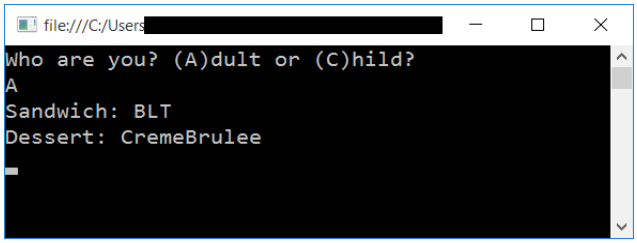
```
/// <summary>
/// A concrete object
/// </summary>
class GrilledCheese : Sandwich { }

/// <summary>
/// A concrete object
/// </summary>
class IceCreamSundae : Dessert { }
```

```
/// <summary>
/// A concrete factory which creates concrete objects by implementing the
abstract factory's methods.
/// </summary>
class KidCuisineFactory : RecipeFactory
{
    public override Sandwich CreateSandwich()
    {
        return new GrilledCheese();
    }

    public override Dessert CreateDessert()
    {
        return new IceCreamSundae();
    }
}
```

How do we use all these classes we've just defined? We implement the **Client** participant!

Let's have our Client ask the user if they are an adult or a child, then display the corresponding menu items.

```
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("Who are you? (A)dult or (C)hild?");
        char input = Console.ReadKey().KeyChar;
        RecipeFactory factory;
        switch(input)
        {
            case 'A':
                factory = new AdultCuisineFactory();
                break;

            case 'C':
                factory = new KidCuisineFactory();
                break;

            default:
                throw new NotImplementedException();

        }

        var sandwich = factory.CreateSandwich();
        var dessert = factory.CreateDessert();

        Console.WriteLine("\nSandwich: " + sandwich.GetType().Name);
        Console.WriteLine("Dessert: " + dessert.GetType().Name);

        Console.ReadKey();
    }
}
```

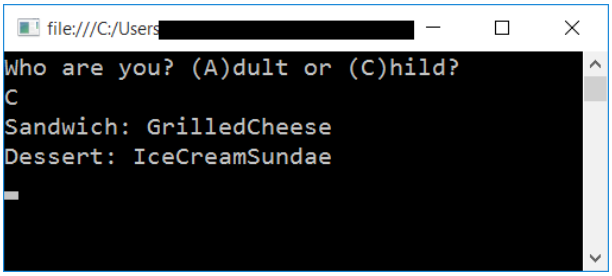


As you can see in the two screenshots on this page, if you specify that you are an adult, the factories return the adult dishes; same if you are a child, they return the child dishes.

Will I Ever Use This Pattern?

Unquestionably. Abstract Factory is an extremely common pattern, and as mentioned earlier it enables architectures such as Dependency Injection.

That said, it's also one of the patterns that's prone to overuse: it's easy to start using Abstract Factories anytime you need to create objects. Be aware of when you decide to use this pattern, and make sure you actually need it.



Summary

The Abstract Factory pattern allows us to generically define families of related objects, leaving the actual concretions for those objects to be implemented as needed. It's best used for scenarios in which you need to create lots of unrelated objects in a manner that allows for maximum code reuse.

Day 3: Façade

What Is This Pattern?

The Façade design pattern is a simple structure laid over a more complex structure.

The idea of the Façade is that if you don't want other code accessing the complex bits of a class or process, you hide those bits by covering them with a Façade.

The Rundown

- **Type:** Structural
- **Useful?** 5/5 (Extremely)
- **Good For:** Hiding complexity which cannot be refactored away.
- **Example Code:** On GitHub
- **Blog Post:** "Facade - The Daily Design Pattern"

The Participants

- The **Subsystems** are any classes or objects which implement functionality but can be "wrapped" or "covered" by the Facade to simplify an interface.
- The **Facade** is the layer of abstraction above the Subsystems, and knows which Subsystem to delegate appropriate work to.

A Delicious Example

To demonstrate how we use the Facade pattern, let's think about a restaurant.

In most kitchens, the work area is divided into sections, such as hot prep, salad prep, the bar, the fountain, etc.

If you are a patron at a restaurant and you sit down at a booth, do you care what part of your meal is made at what section of the restaurant? Of course not. **There is naturally a layer of abstraction in place: the server.**

The server knows where to place each order and where to pick those parts of the order up from. We'll model this relationship to demonstrate how the Façade pattern can simplify the structure of our code.

First, let's create a class for the restaurant patron:

```
/// <summary>
/// Patron of the restaurant
/// </summary>
class Patron
{
    private string _name;

    public Patron(string name)
    {
        this._name = name;
    }

    public string Name
    {
        get { return _name; }
    }
}
```

Let's also define a base class representing all food items sold at this restaurant; an interface representing all sections of this restaurant's kitchen; and a class representing a patron's order:

```
/// <summary>
```

```
/// All items sold in the restaurant must inherit from this.
/// </summary>
class FoodItem { public int DishID; }

/// <summary>
/// Each section of the kitchen must implement this interface.
/// </summary>
interface KitchenSection
{
    FoodItem PrepDish(int DishID);
}

/// <summary>
/// Orders placed by Patrons.
/// </summary>
class Order
{
    public FoodItem Appetizer { get; set; }
    public FoodItem Entree { get; set; }
    public FoodItem Drink { get; set; }
}
```

Now we can start to model the sections of the kitchen, AKA the **Subsystem** participants. Here's the classes for `ColdPrep`, `HotPrep`, and `Bar`:

```
/// <summary>
/// A division of the kitchen.
/// </summary>
class ColdPrep : KitchenSection
{
    public FoodItem PrepDish(int dishID)
    {
        //Go prep the cold item
        return new FoodItem()
        {
            DishID = dishID
        };
    }
}

/// <summary>
/// A division of the kitchen.
/// </summary>
class HotPrep : KitchenSection
{
    public FoodItem PrepDish(int dishID)
    {
        //Go prep the hot entree
        return new FoodItem()
        {
            DishID = dishID
        };
    }
}
```

```
/// <summary>
/// A division of the kitchen.
/// </summary>
class Bar : KitchenSection
{
    public FoodItem PrepDish(int dishID)
    {
        //Go mix the drink
        return new FoodItem()
        {
            DishID = dishID
        };
    }
}
```

Finally, we need the actual **Façade** participant, which is our `Server` class:

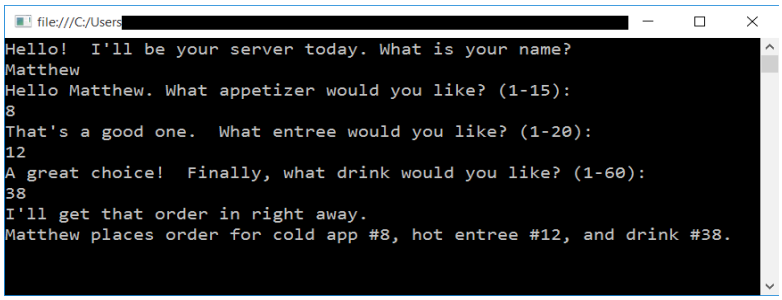
```
/// <summary>
/// The actual "Facade" class, which hides the complexity of the KitchenSection
/// classes. After all, there's no reason a patron should order each part of
/// their meal individually.
/// </summary>
class Server
{
    private ColdPrep _coldPrep = new ColdPrep();
    private Bar _bar = new Bar();
    private HotPrep _hotPrep = new HotPrep();

    public Order PlaceOrder(Patron patron,
                            int coldAppID,
                            int hotEntreeID,
                            int drinkID)
    {
        Console.WriteLine("{0} places order for cold app #"
                          + coldAppID.ToString()
                          + ", hot entree #" + hotEntreeID.ToString()
                          + ", and drink #" + drinkID.ToString() + ".");

        Order order = new Order();

        order.Appetizer = _coldPrep.PrepDish(coldAppID);
        order.Entree = _hotPrep.PrepDish(hotEntreeID);
        order.Drink = _bar.PrepDish(drinkID);

        return order;
    }
}
```

With all of these in place, we can use the Main method to show how a patron might place an order and how the server (the Facade) would direct the appropriate pieces of that order to the kitchen sections (the Subsystems):

```
static void Main(string[] args)
{
```

```
    Server server = new Server();

    Console.WriteLine("Hello!  I'll be your server today. What is your name?");
    var name = Console.ReadLine();

    Patron patron = new Patron(name);

    Console.WriteLine("Hello " + patron.Name
                      + ". What appetizer would you like? (1-15):");
    var appID = int.Parse(Console.ReadLine());

    Console.WriteLine("That's a good one.  What entree would you like? (1-
20):");
    var entreeID = int.Parse(Console.ReadLine());

    Console.WriteLine("A great choice!  Finally, what drink would you like? (1-
60):");
    var drinkID = int.Parse(Console.ReadLine());

    Console.WriteLine("I'll get that order in right away.");

    //Here's what the Facade simplifies
    server.PlaceOrder(patron, appID, entreeID, drinkID);

    Console.ReadKey();
}
```

Will I Ever Use This Pattern?



All the damn time. Seriously, the Façade pattern is so general that it applies to almost every major app I've worked on, especially those where I couldn't refactor or modify pieces of said apps for various reasons. You'll probably be using it a lot, even when you might not notice that a Façade pattern is being applied.

Summary

The Façade pattern is a simple (or at least *simpler*) overlay on top of a group of more complex subsystems. The Façade knows which Subsystem to direct different kinds of work toward. And it is really, really common, so it's one of the patterns we should know thoroughly.

Day 4: Adapter

What Is This Pattern?

The Adapter design pattern attempts to reconcile the differences between two otherwise-incompatible interfaces. This pattern is especially useful when attempting to adapt to an interface which cannot be refactored (e.g. when a particular interface is controlled by a web service or API).

The Rundown

- **Type:** Structural
- **Useful?** 4/5 (Very)
- **Good For:** Adapting two interfaces together when one or more of those interfaces cannot be refactored.
- **Example Code:** On GitHub
- **Blog Post:** "Adapter - The Daily Design Pattern"

The Participants

- The **Target** defines the domain-specific interface in use by the Client.
- The **Client** collaborates with objects which conform to the Target.
- The **Adapter** adapts the Adaptee to the Target.
- The **Adaptee** is the interface that needs adapting (i.e. the one that cannot be refactored or changed).

A Delicious Example

Vegetarians beware; for the Adapter design pattern example, we're gonna cook some meat. *Lots* of it.

Let's imagine that we maintain a meat safe-cooking temperature database. The US Food & Drug Administration maintains a list of temperatures to which meat must be cooked before it is safe for human consumption. We're going to show how the Adapter design pattern can be used to adapt an old, unchangeable API to a new object-oriented system.

First, we need to define the old, creaky, ancient API that we want to adapt (this is the **Adaptee** participant).

```
public enum TemperatureType
{
```

```
    Fahrenheit,
    Celsius
}
/// <summary>
/// The legacy API which must be converted to the new structure
/// </summary>
class MeatDatabase
{
    public float GetSafeCookTemp(string meat, TemperatureType tempType)
    {
        if (tempType == TemperatureType.Fahrenheit)
        {
            switch (meat)
            {
                case "beef":
                case "pork":
                    return 145f;

                case "chicken":
                case "turkey":
                    return 165f;

                default:
                    return 165f;
            }
        }
        else
        {
            switch (meat)
            {
                case "beef":
                case "veal":
                case "pork":
                    return 63f;

                case "chicken":
                case "turkey":
                    return 74f;

                default:
                    return 74f;
            }
        }
    }

    public int GetCaloriesPerOunce(string meat)
    {
        switch (meat.ToLower())
        {
            case "beef": return 71;
            case "pork": return 69;
            case "chicken": return 66;
            case "turkey": return 38; //Wow, turkey is lean!
            default: return 0;
```

```
        }
    }

    public double GetProteinPerOunce(string meat)
    {
        switch (meat.ToLower())
        {
            case "beef": return 7.33f;
            case "pork": return 7.67f;
            case "chicken": return 8.57f;
            case "turkey": return 8.5f;
            default: return 0d;
        }
    }
}
```

This legacy API does not properly model objects in an object-oriented fashion. Where this API returns results from methods, we know that that data (safe cook temperature, calories per ounce, protein per ounce) should really be properties in some kind of `Meat` object.

So, let's create that Meat object (which is our **Target** participant):

```
/// <summary>
/// The new Meat class, which represents details about a specific kind of meat.
/// </summary>
class Meat
{
    protected string MeatName;
    protected float SafeCookTempFahrenheit;
    protected float SafeCookTempCelsius;
    protected double CaloriesPerOunce;
    protected double ProteinPerOunce;

    // Constructor
    public Meat(string meat)
    {
        this.MeatName = meat;
    }

    public virtual void LoadData()
    {
        Console.WriteLine("\nMeat: {0} ------ ", MeatName);
    }
}
```

Problem is, we cannot modify the legacy API. So how are we supposed to take data from the API, where it is returned from method calls, and instead have that data modeled as properties of an object?

This is where our **Adapter** participant comes into play: we need another class that inherits from `Meat` but maintains a reference to the API such that the API's data can be loaded into an instance of the `Meat` class:

```
/// <summary>
```

```
/// The Adapter class, which wraps the Meat class.
/// </summary>
class MeatDetails : Meat
{
    private MeatDatabase _meatDatabase;

    public MeatDetails(string name)
        : base(name) { }

    public override void LoadData()
    {
        // The Adaptee
        _meatDatabase = new MeatDatabase();

        SafeCookTempFahrenheit = _meatDatabase.GetSafeCookTemp(MeatName,
TemperatureType.Fahrenheit);
        SafeCookTempCelsius = _meatDatabase.GetSafeCookTemp(MeatName,
TemperatureType.Celsius);
        CaloriesPerOunce = _meatDatabase.GetCaloriesPerOunce(MeatName);
        ProteinPerOunce = _meatDatabase.GetProteinPerOunce(MeatName);

        base.LoadData();
        Console.WriteLine(" Safe Cook Temp (F): {0}", SafeCookTempFahrenheit);
        Console.WriteLine(" Safe Cook Temp (C): {0}", SafeCookTempCelsius);
        Console.WriteLine(" Calories per Ounce: {0}", CaloriesPerOunce);
        Console.WriteLine(" Protein per Ounce: {0}", ProteinPerOunce);
    }
}
```

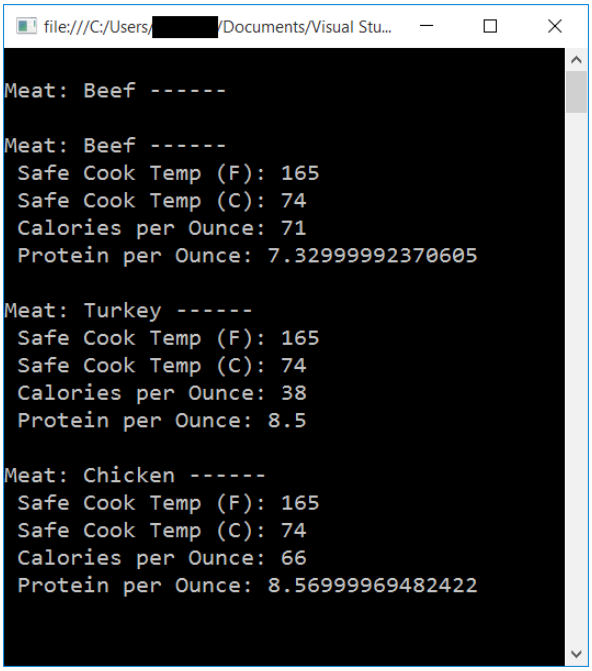
Finally, we can write the Main method

```
static void Main(string[] args)
{
    //Non-adapted
    Meat unknown = new Meat("Beef");
    unknown.LoadData();

    //Adapted
    MeatDetails beef = new MeatDetails("Beef");
    beef.LoadData();

    MeatDetails turkey = new MeatDetails("Turkey");
    turkey.LoadData();

    MeatDetails chicken = new MeatDetails("Chicken");
    chicken.LoadData();

    Console.ReadKey();
}
```

Each `LoadData` call from an adapted class loads the data from the legacy API into that instance. You can see a screenshot of the sample application's output on the next page.

Will I Ever Use This Pattern?

Most likely. As I've been mentioning, the pattern is extremely useful when you're trying to adapt old or legacy systems to new designs, so if you're ever in that situation the Adapter pattern might be the best fit for your project.



Summary

The Adapter pattern attempts to reconcile two incompatible interfaces, and is especially useful when one or both of those interfaces cannot be refactored.

Day 4.5: Adapter vs Façade

The Adapter pattern and the Façade pattern are very similar patterns, but they are used slightly differently.

- Façade creates a new interface; Adapter re-uses an existing one.
- Façade hides several interfaces; Adapter makes two existing ones work together.
- Façade is the equivalent of saying "This is never gonna work; I will build my own."; Adapter is the equivalent of "Of course it will work, it just needs a little tweaking."

The key to remember about these two patterns is this:

Use Adapter for when you're adapting one-to-one subsystems where at least one of them cannot be refactored.

Use Façade for when you must hide complexity from three or more subsystems where at least one cannot be refactored.

Day 5: Bridge

What Is This Pattern?

The Bridge pattern seeks to decouple an abstraction from its implementation such that both can vary independently. Effectively, the Bridge maintains a reference to both abstraction and implemention but doesn't implement either, thereby allowing the details of both to remain in their separate classes.

In object-oriented programming, the concept of inheritance is crucial to developing objects. This binds the implementation to its abstraction, and often that's exactly what we want.

However, there can be scenarios in which one class inheriting from another might not be the best solution, particularly when multiple inheritances can be used. Into this void steps the Bridge design pattern.

This pattern is especially useful for scenarios in which changes to the implementation of an object should have no bearing on how their clients use said implementations. Bridge also differs from Adapter in that Bridge is used when designing new systems while Adapter is used to adapt old systems to new ones.

The Rundown

- **Type:** Structural
- **Useful?** 3/5 (Sometimes)
- **Good For:** Allowing lots of variation between implementations of interfaces.
- **Example Code:** On GitHub
- **Blog Post:** "Bridge - The Daily Design Pattern"

The Participants

- The **Abstraction** defines an interface and maintains a reference to an Implementer.
- The **RefinedAbstraction** extends the interface defined by the Abstraction.
- The **Implementer** defines the interface for the ConcreteImplementer objects. This interface does not need to correspond to the Abstraction's interface.
- The **ConcreteImplementer** objects implement the Implementer interface.

Background

In real life, my brother has Celiac disease, a condition in which his body cannot properly process gluten. Because of this, he cannot eat wheat, rye, barley, oats, or anything made from any of those ingredients; if he does, he's probably going to be unwillingly asleep for the next six hours or so and could cause permanent damage to his digestive system.

Consequently it can be difficult for him to order a meal from restaurants, since often they don't provide the proper special-needs meal he needs (and even if they do, the environment in which the food is prepped is often not properly ventilated or sterilized, making cross-contamination likely).

In his honor, let's model a system by which we can order various

special-needs meals from many different restaurants.

The idea goes like this: I should be able to pick a type of special meal and pick a restaurant, without needing to know exactly what either of those things are (e.g. a dairy-free meal from a diner or a gluten-free meal from a fancy restaurant).

In a traditional inheritance model, we might have the following classes:

```
interface IOrder {}
class DairyFreeOrder : IOrder {}
class GlutenFreeOrder : IOrder {}
```

But what if we also need to keep track of what kind of restaurant the order came from? This is orthogonal to what the meal *is*, but is still a part of the model. In this case, we might end up with a crazy inheritance tree:

```
interface IOrder {}
class DairyFreeOrder : IOrder {}
class GlutenFreeOrder : IOrder {}
interface IDinerOrder : IOrder {}
class DinerDairyFreeOrder : DairyFreeOrder, IDinerOrder {}
class DinerGlutenFreeOrder : GlutenFreeOrder, IDinerOrder {}
interface IFancyRestaurantOrder : IOrder {}
class FancyRestaurantDairyFreeOrder : DairyFreeOrder, IFancyRestaurantOrder {}
class FancyRestaurantGlutenFreeOrder : GlutenFreeOrder, IFancyRestaurantOrder {}
```

So we're only modeling two orthogonal properties (dairy-free vs gluten-free and diner vs fancy restaurant) but we need three interfaces and six classes? Seems like overkill, don't you think?

The Bridge design pattern seeks to divide the responsibility of these interfaces such that they're much more reusable. What we want is to end up with something like the next example:

```
interface IOrder { }

//This class keeps a private reference to an IRestaurantOrder
class DairyFreeOrder : IOrder { }

//This class also keeps a private reference to an IRestaurantOrder
class GlutenFreeOrder : IOrder { }

interface IRestaurantOrder : IOrder { }
class DinerOrder : IRestaurantOrder { }
class FancyRestaurantOrder : IRestaurantOrder { }
```

Let's expand this example to fully create the Bridge design pattern.

A Delicious Example

To implement the Bridge design pattern correctly and model our special-needs ordering system, we must first write our **Implementer** participant, which will define a method for

placing an order:

```
/// <summary>
/// Implementer which defines an interface for placing an order
/// </summary>
public interface IOrderingSystem
{
    void Place(string order);
}
```

We also need the **Abstraction** participant, which for this day's demo is an abstract class which will define a method for sending an order *and* keep a reference to the Implementer:

```
/// <summary>
/// Abstraction which represents the sent order and maintains a reference to the
/// restaurant where the order is going.
/// </summary>
public abstract class SendOrder
{
    //Reference to the Implementer
    public IOrderingSystem _restaurant;

    public abstract void Send();
}
```

Now we can start defining our **RefinedAbstraction** classes. For this demo, let's take those two kinds of special-needs meals from earlier (dairy-free and gluten-free) and implement RefinedAbstraction objects for them.

```
/// <summary>
/// RefinedAbstraction for a dairy-free order
/// </summary>
public class SendDairyFreeOrder : SendOrder
{
    public override void Send()
    {
        _restaurant.Place("Dairy-Free Order");
    }
}

/// <summary>
/// RefinedAbstraction for a gluten free order
/// </summary>
public class SendGlutenFreeOrder : SendOrder
{
    public override void Send()
    {
        _restaurant.Place("Gluten-Free Order");
    }
}
```

The final piece is to define the ordering systems for the different types of restaurants (which are our **ConcreteImplementer** participants):

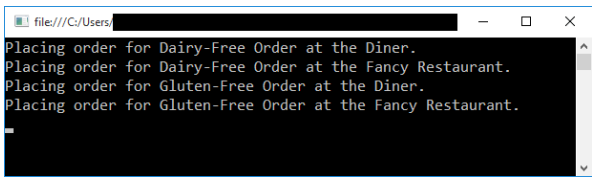
```
/// <summary>
/// ConcreteImplementer for an ordering system at a diner.
/// </summary>
public class DinerOrders : IOrderingSystem
{
    public void Place(string order)
    {
        Console.WriteLine("Placing order for " + order + " at the Diner.");
    }
}

/// <summary>
/// ConcreteImplementer for an ordering system at a fancy restaurant.
/// </summary>
public class FancyRestaurantOrders : IOrderingSystem
{
    public void Place(string order)
    {
        Console.WriteLine("Placing order for " + order
                          + " at the Fancy Restaurant.");
    }
}
```

To demonstrate how this works, let's create a Main method which uses the Bridge to create various orders and send them to different restaurants.

```
static void Main(string[] args)
{
    SendOrder _sendOrder = new SendDairyFreeOrder();
    _sendOrder._restaurant = new DinerOrders();
    _sendOrder.Send();

    _sendOrder._restaurant = new FancyRestaurantOrders();
    _sendOrder.Send();

    _sendOrder = new SendGlutenFreeOrder();
    _sendOrder._restaurant = new DinerOrders();
    _sendOrder.Send();

    _sendOrder._restaurant = new FancyRestaurantOrders();
    _sendOrder.Send();

    Console.ReadKey();
}
```

The key part of the Bridge pattern is that each side (Abstraction and Implementer) can now change independently, and neither system will care.

If you run the app, you will find that you can send any order to any restaurant. Further, if any of the abstractions (the orders) change their definition, the implementers don't actually care; and vice-versa, if the implementers change their



implementation, the abstractions don't need to change as well.

Will I Ever Use This Pattern?

Probably. As mentioned above, this pattern is very useful when designing systems where multiple different kinds of inheritance are possible; Bridge allows you to implement these inheritances without tightly binding to their abstractions.

That said, this is one of those patterns where the complexity needed to implement it may well cancel out its benefits. Don't forget to think critically about your situation and determine if you really need a pattern before you start refactoring toward it!

Summary

The Bridge design pattern seeks to allow abstractions and implementations to vary independently. This becomes useful when dealing with situations where regular inheritance would cause our projects to have too many or too complex inheritance trees. But, the Bridge pattern invokes a considerable amount of complexity, so be sure that it solves your particular problem before starting to refactor toward it!

Day 6: Template Method

What Is This Pattern?

The Template Method design pattern defines the outline or skeleton of an operation, but leaves the specific steps involved to be defined by subclasses.

In other words, the Template Method pattern defines in what order certain steps should occur, but can optionally leave the specific details of those steps to be implemented by other classes. Whereas Factory Method did something similar with creating objects, Template Method does this for the *behavior* of those objects.

The Rundown

- **Type:** Behavioral
- **Useful?** 4/5 (Very, with some caveats)
- **Good For:** Creating an outline of an algorithm but letting specific steps be implemented by other classes.
- **Example Code:** On GitHub
- **Blog Post:** "Template Method - The Daily Design Pattern"

The Participants

- The **AbstractClass** defines a set of abstract operations which can (optionally) be implemented by ConcreteClass objects. It also implements a template method which controls the order in which those abstract operations occur.
- The **ConcreteClass** objects implement the operations defined by the AbstractClass.

A Delicious Example

To properly demo this design pattern, let's talk about something humanity has been doing for 30,000 years: **baking bread**.

There are easily hundreds of types of bread currently being made in the world, but each kind involves specific steps in order to make them. While acknowledging that this doesn't necessarily cover all kinds of bread that are possible to make, let's say that there are three basic steps in making bread:

1. Mix the ingredients together
2. Bake the mixture
3. Slice the resulting bread

We want to model a few different kinds of bread that all use this same pattern, which (no surprise) is a good fit for the Template Method design pattern.

First, let's create an **AbstractClass** `Bread` which represents all breads we can bake:

```
/// <summary>
/// The AbstractClass participant which contains the template method.
/// </summary>
abstract class Bread
{
    public abstract void MixIngredients();

    public abstract void Bake();

    public virtual void Slice()
    {
        Console.WriteLine("Slicing the " + GetType().Name + " bread!");
    }

    // The template method
    public void Make()
    {
        MixIngredients();
        Bake();
        Slice();
    }
}
```

Notice that the `MixIngredients()` and `Bake()` methods are abstract, while the `Slice()` method is virtual. This is intentional: the method by which you slice bread is not likely to change depending on the kind of bread you make. Further, the `Make()` method is the Template Method that gives this pattern its name.

Let's extend this example by implementing several **ConcreteClass** objects for different types of bread:

```
class TwelveGrain : Bread
{
    public override void MixIngredients()
    {
        Console.WriteLine("Gathering Ingredients for 12-Grain Bread.");
    }

    public override void Bake()
    {
        Console.WriteLine("Baking the 12-Grain Bread. (25 minutes)");
    }
}

class Sourdough : Bread
{
    public override void MixIngredients()
    {
        Console.WriteLine("Gathering Ingredients for Sourdough Bread.");
    }

    public override void Bake()
    {
        Console.WriteLine("Baking the Sourdough Bread. (20 minutes)");
    }
}

class WholeWheat : Bread
{
    public override void MixIngredients()
    {
        Console.WriteLine("Gathering Ingredients for Whole Wheat Bread.");
    }

    public override void Bake()
    {
        Console.WriteLine("Baking the Whole Wheat Bread. (15 minutes)");
    }
}
```

Once we've defined a few types of bread, we can simulate making them in our `Main()` method, like so:

```
static void Main(string[] args)
{
    Sourdough sourdough = new Sourdough();
    sourdough.Make();
```

```
    TwelveGrain twelveGrain = new TwelveGrain();
    twelveGrain.Make();

    WholeWheat wholeWheat = new WholeWheat();
    wholeWheat.Make();

    Console.ReadKey();
}
```

It's just that simple. In fact, Template Method is (arguably) the simplest and most flexible of all the behavioral design patterns.

Will I Ever Use This Pattern?

Almost certainly. I'd be willing to bet that most of you dear readers have already used this pattern and may not have known what it was called. This pattern is extremely common, flexible, and useful for many different applications and scenarios.

...But. It's not without problems. Jimmy Bogard explains:

> "While some gravitate towards the Singleton pattern to abuse after they learn the GoF patterns, that wasn't the case for me. Instead, I fell in love with the Template [Method] Pattern. But there's a problem with [this] pattern as the golden hammer for every incidence of duplication we find in our application. **The Template Method favors inheritance over composition**."

And, to be fair, he's right. Template Method forces a class to inherit from a class rather than promoting object composition. If we're looking for strict-object-oriented design, Template Method could be better replaced by other patterns we've yet to cover, such as Strategy or Command.

But I'm not willing to go as far as saying "don't use Template Method." As with all the other patterns, their applications depend on what problem you need to solve and how you want to do so. Template Method is prone to over-use, so be careful with it.

Summary

The Template Method design pattern allows for an object to set up a skeleton of an algorithm but leave the implementation details up to the concrete classes to implement. It is a fairly simple pattern, so you're very likely to utilize it at some point, but as with all patterns try to be aware of why you are using a particular pattern and what problem you are attempting to solve with it.

Day 7: Iterator

What Is This Pattern?

The Iterator pattern provides a way to access objects in an underlying representation without exposing access to the representation itself.

The idea is that we'll have a class (the "Iterator") which contains a reference to a corresponding aggregate object, and that Iterator can traverse over its aggregate to retrieve individual objects.

If, like myself, you primarily work in the .NET/ASP.NET world and have used LINQ, then you have already used many implementations of this pattern.

The Rundown

- **Type:** Behavioral
- **Useful?** 5/5 (Extremely)
- **Good For:** Extracting objects from a collection without exposing the collection itself.
- **Example Code:** On GitHub
- **Blog Post:** "Iterator - The Daily Design Pattern"

The Participants

- The **Iterator** defines an interface for accessing an Aggregate object and traversing elements within that Aggregate.
- The **ConcreteIterator** implements the Iterator interface and keeps track of its current position within the Aggregate.
- The **Aggregate** defines an interface for creating an Iterator object.
- The **ConcreteAggregate** implements the Iterator creation interface and returns a ConcreteIterator for that ConcreteAggregate.

A Delicious Example

To demo how we might use the Iterator design pattern, let's talk about my favorite sugary snack: jelly beans. So far as I am concerned, these little nuggets of sugar and flavor are the best thing since sliced bread.

To properly demo how we might use the Iterator pattern, let's build a collection for a group of jelly beans and have that

collection create an iterator for itself. To do this, we must first define a class to represent a single jelly bean.

```
/// <summary>
/// Our collection item.  Mostly because I'm a sucker for jelly beans.
/// </summary>
class JellyBean
{
    private string _flavor;

    // Constructor
    public JellyBean(string flavor)
    {
        this._flavor = flavor;
    }


    public string Flavor
    {
        get { return _flavor; }
    }
}
```

Next we need both our **Aggregate** and **ConcreteAggregate** participants, which represent a collection of jelly beans.

```
/// <summary>
/// The aggregate interface
/// </summary>
interface ICandyCollection
{
    Iterator CreateIterator();
}

/// <summary>
/// The ConcreteAggregate class
/// </summary>
class JellyBeanCollection : ICandyCollection
{
    private ArrayList _items = new ArrayList();

    public Iterator CreateIterator()
    {
        return new Iterator(this);
    }

    // Gets jelly bean count
    public int Count
    {
        get { return _items.Count; }
    }

    // Indexer
    public object this[int index]
```

```
    {
        get { return _items[index]; }
        set { _items.Add(value); }
    }
}
```

Now that we have our collection item (`JellyBean`) and our collection (`JellyBeanCollection`) defined, we must implement our Iterator and ConcreteIterator participants.

The **Iterator** participant will be an interface which each ConcreteIterator must implement:

```
/// <summary>
/// The 'Iterator' interface
/// </summary>
interface IJellyBeanIterator
{
    JellyBean First();
    JellyBean Next();
    bool IsDone { get; }
    JellyBean CurrentBean { get; }
}
```

The **ConcreteIterator** will implement the methods defined by the Iterator interface:

```
/// <summary>
/// The 'ConcreteIterator' class
/// </summary>
class JellyBeanIterator : IJellyBeanIterator
{
    private JellyBeanCollection _jellyBeans;
    private int _current = 0;
    private int _step = 1;

    // Constructor
    public JellyBeanIterator(JellyBeanCollection beans)
    {
        this._jellyBeans = beans;
    }

    // Gets first jelly bean
    public JellyBean First()
    {
        _current = 0;
        return _jellyBeans[_current] as JellyBean;
    }

    // Gets next jelly bean
    public JellyBean Next()
    {
        _current += _step;
        if (!IsDone)
            return _jellyBeans[_current] as JellyBean;
        else
            return null;
```

```
    }

    // Gets current iterator candy
    public JellyBean CurrentBean
    {
        get { return _jellyBeans[_current] as JellyBean; }
    }

    // Gets whether iteration is complete
    public bool IsDone
    {
        get { return _current >= _jellyBeans.Count; }
    }
}
```

Notice that the ConcreteAggregate needs to implement methods by which we can manipulate objects within the collection, without exposing the collection itself. This is how our example fits with the Iterator design pattern.

Finally, in our Main method, we can create a collection of jelly beans and then iterate over them:

```
static void Main(string[] args)
{
    // Build a collection of jelly beans
    JellyBeanCollection collection = new JellyBeanCollection();
    collection[0] = new JellyBean("Cherry");
    collection[1] = new JellyBean("Bubble Gum");
    collection[2] = new JellyBean("Root Beer");
    collection[3] = new JellyBean("French Vanilla");
    collection[4] = new JellyBean("Licorice");
    collection[5] = new JellyBean("Buttered Popcorn");
    collection[6] = new JellyBean("Juicy Pear");
    collection[7] = new JellyBean("Cinnamon");
    collection[8] = new JellyBean("Coconut");

    // Create iterator
    JellyBeanIterator iterator = collection.CreateIterator();

    Console.WriteLine("Gimme all the jelly beans!");

    for (JellyBean item = iterator.First();
        !iterator.IsDone; item = iterator.Next())
    {
        Console.WriteLine(item.Flavor);
    }

    Console.ReadKey();
}
```

Will I Ever Use This Pattern?

Absolutely. The pattern is astonishingly useful when attempting to retrieve objects from collections that you'd rather not expose to outside usage (because that's, like, the pattern's *entire purpose*). It's so common that many frameworks (including ASP.NET) support it natively in their designs.

Summary

The Iterator pattern provides a manner in which we can access and manipulate objects in a collection without exposing the collection itself. This pattern has the rare luxury of being both incredibly common and incredibly useful, so keep it in mind; once you know what it is, you'll start seeing it everywhere.

Day 8: Observer

What Is This Pattern?

The Observer pattern seeks to allow objects to notify their observers when their internal state changes.

This means that a single object will need to be aware of the objects that observe it, and need to be able to communicate to those observers that the subject's state changed. Further, the observers should be notified automatically.

The Rundown

- **Type:** Behavioral
- **Useful?** 4/5 (Very)
- **Good For:** Notifying observer objects that a particular subject's state changed.
- **Example Code:** On GitHub
- **Blog Post:** "Observer - The Daily Design Pattern"

The Participants

- The **Subject** knows its Observers and provides an interface for attaching or detaching any number of Observer objects.
- The **ConcreteSubject** objects store the states of interest to the Observers and are responsible for sending a notification when the ConcreteSubject's state changes.
- The **Observer** defines an updating interface for objects that should be notified of changes in a Subject.
- The **ConcreteObserver** objects maintain a reference to a ConcreteSubject and implement the Observer updating interface to keep its state consistent with that of the

Subject's.

A Delicious Example

To understand how we might use the Observer design pattern in the real world, let's imagine that we need a system to model the fluctuating prices of vegetables at our local market.

On some days, the vegetables will be more expensive than on other days, due to factors like the size of the harvest or the size of the vegetables themselves.

Further, we need to allow restaurants to watch the prices and place an order when the price for a particular vegetable falls below a specified threshold, which is different for each restaurant.

In short, we need to build a system in which restaurants are notified by changing vegetable prices. We can start to do this by defining our **Subject** participant, which will be an abstract class representing a certain vegetable. This class must be able to attach or detach observers and keep track of its own price. Here's the sample abstract class:

```
/// <summary>
/// The Subject abstract class
/// </summary>
abstract class Veggies
{
    private double _pricePerPound;
    private List<IRestaurant> _restaurants = new List<IRestaurant>();

    public Veggies(double pricePerPound)
    {
        _pricePerPound = pricePerPound;
    }

    public void Attach(IRestaurant restaurant)
    {
        _restaurants.Add(restaurant);
    }

    public void Detach(IRestaurant restaurant)
    {
        _restaurants.Remove(restaurant);
    }

    public void Notify()
    {
        foreach (IRestaurant restaurant in _restaurants)
        {
```

```
            restaurant.Update(this);
        }

        Console.WriteLine("");
    }

    public double PricePerPound
    {
        get { return _pricePerPound; }
        set
        {
            if (_pricePerPound != value)
            {
                _pricePerPound = value;
                Notify(); //Automatically notify our observers of price changes
            }
        }
    }
}
```

Once we have the Subject defined, we need a **ConcreteSubject** to represent a specific vegetable; in this case, we'll call it `Carrots`:

```
/// <summary>
/// The ConcreteSubject class
/// </summary>
class Carrots : Veggies
{
    public Carrots(double price) : base(price) { }
}
```

Now we can define our **Observer** participant. Remember that restaurants want to observe the vegetable prices, so our Observer will naturally be an interface `IRestaurant`, and this interface must define a method by which its implementers can be updated:

```
/// <summary>
/// The Observer interface
/// </summary>
interface IRestaurant
{
    void Update(Veggies veggies);
}
```

Finally we need our ConcreteObserver class, which represent specific restaurants. This class must implement the `Update()` method from `IRestaurant`:

```
/// <summary>
/// The ConcreteObserver class
/// </summary>
class Restaurant : IRestaurant {
    private string _name;
    private Veggies _veggie;
    private double _purchaseThreshold;
```

```
    public Restaurant(string name, double purchaseThreshold)
    {
        _name = name;
        _purchaseThreshold = purchaseThreshold;
    }

    public void Update(Veggies veggie)
    {
        Console.WriteLine(
            "Notified {0} of {1}'s " + " price change to {2:C} per pound.",
            _name, veggie.GetType().Name, veggie.PricePerPound);
        if(veggie.PricePerPound < _purchaseThreshold)
        {
            Console.WriteLine(_name + " wants to buy some "
                              + veggie.GetType().Name + "!");
        }
    }
}
```

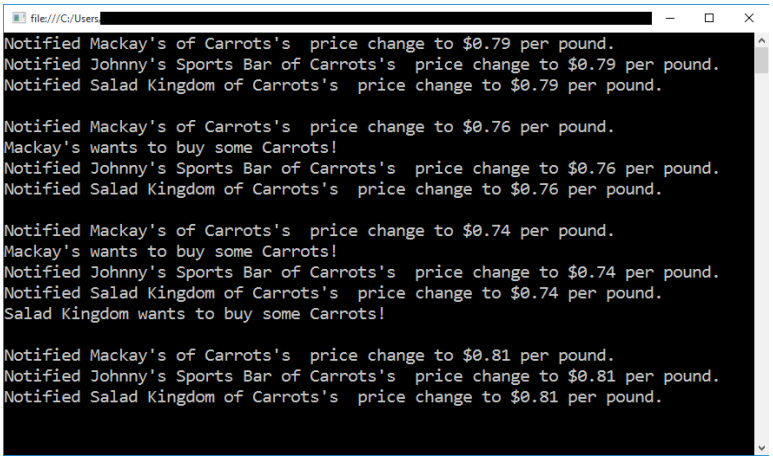
Note that the Restaurants will want to buy veggies if the price dips below a certain threshold amount, which differs per restaurant.

To put this all together, in our Main method we can define a few restaurants that want to observe the price of carrots, then fluctuate that price:

```
static void Main(string[] args)
{
    // Create price watch for Carrots and attach restaurants
    // that buy carrots from suppliers.
    Carrots carrots = new Carrots(0.82);
    carrots.Attach(new Restaurant("Mackay's", 0.77));
    carrots.Attach(new Restaurant("Johnny's Sports Bar", 0.74));
    carrots.Attach(new Restaurant("Salad Kingdom", 0.75));

    // Fluctuating carrot prices will notify subscribing restaurants.
    carrots.PricePerPound = 0.79;
    carrots.PricePerPound = 0.76;
    carrots.PricePerPound = 0.74;
    carrots.PricePerPound = 0.81;

    Console.ReadKey();
}
```

If we run the app, we will see that as the price changes, the restaurants get notified, and if the price drops below each restaurant's threshold, that restaurant then wants to place an order. The screenshot on the next page shows what the output from the sample app would look like.

Looking at the screenshot on this page, you'll notice that the subject object (`Carrots`) automatically notifies the observer restaurants of its own price changes, which can then decide what to do with that information (e.g. place an order for more carrots).



Will I Ever Use This Pattern?

Most likely. This is a fairly common pattern, and the ability to automatically notify dependent objects of a subject's state change is highly desirable in my opinion. However, as with all software design patterns, be sure you aren't shoehorning the Observer design pattern into a solution where it doesn't fit.

Summary

The Observer design pattern seeks to allow Observer objects to automatically receive notifications (and possibly change their own state) when a Subject class changes its state. In short, should the Subject's internal state change, its Observers will be notified of said change.

Day 9: Memento

What Is This Pattern?

The Memento pattern seeks to capture and externalize and object's state so that the object can be restored to this state at a later time.

The purpose of this pattern is to separate the current state of the object from a previous state, so that if something happens to the current state (it gets corrupted, it gets lost, it tries to secede from the Union) the object's state can be restored from its Memento (whether via a civil war or other, less interesting methods).

For example, let's create a memento of my current state: hungry.

```
var memento = someWriter.CreateMemento(); //Hungry
someWriter.startWriting(); //Change state to writing
```

Now, we can use that memento instance later to restore the state of the someBlogger object.

The Rundown

- **Type:** Structural
- **Useful?** 2/5 (Probably not)
- **Good For:** Restoring an object's state from a previous state by creating a memento of said previous state.
- **Example Code:** On GitHub
- **Blog Post:** "Memento - The Daily Design Pattern"

The Participants

- The **Memento** stores internal state of the Originator object. The Memento has no limit on what it may or may not store (e.g. as much or as little of the Originator's state as needed).
- The **Originator** creates a Memento containing a "snapshot" of its internal state, and then later uses that memento to restore its internal state.
- The **Caretaker** is responsible for the Memento's safekeeping, but does not operate on or examine the contents of that Memento.

A Delicious Example

Well, in so far as "delicious" means "fitting the food theme" but anyway.

Let's imagine a system in which a restaurant needs to record information about the suppliers that bring them their ingredients. For example, a really high-end restaurant might order directly from a local farm, and the restaurant needs to keep track of which ingredients come from which suppliers.

In our system, we need to keep track of how much information we enter about a particular supplier, and be able to restore that information to a previous state if we, say, accidentally enter the wrong address. We can demo this using the Memento pattern.

First, let's create our **Originator** participant, which will create and use Mementos:

```
/// <summary>
/// The Originator class, which is the
/// class for which we want to save
/// Mementos for its state.
/// </summary>
class FoodSupplier
{
    private string _name;
```

```
    private string _phone;
    private string _address;

    public string Name
    {
        get { return _name; }
        set
        {
            _name = value;
            Console.WriteLine("Proprietor:  " + _name);
        }
    }

    public string Phone
    {
        get { return _phone; }
        set
        {
            _phone = value;
            Console.WriteLine("Phone Number: " + _phone);
        }
    }

    public string Address
    {
        get { return _address; }
        set
        {
            _address = value;
            Console.WriteLine("Address: " + _address);
        }
    }

    public FoodSupplierMemento SaveMemento()
    {
        Console.WriteLine("\nSaving current state\n");
        return new FoodSupplierMemento(_name, _phone, _address);
    }

    public void RestoreMemento(FoodSupplierMemento memento)
    {
        Console.WriteLine("\nRestoring previous state\n");
        Name = memento.Name;
        Phone = memento.PhoneNumber;
        Address = memento.Address;
    }
}
```

We also need a **Memento** participant, which is the `FoodSupplierMemento` object used by `FoodSupplier`:

```
/// <summary>
/// The Memento class
/// </summary>
```

```
class FoodSupplierMemento
{
    public string Name { get; set; }
    public string PhoneNumber { get; set; }
    public string Address { get; set; }

    public FoodSupplierMemento(string name, string phone, string address)
    {
        Name = name;
        PhoneNumber = phone;
        Address = address;
    }
}
```

Finally, we need our **Caretaker** participant, which stores the Mementos but never inspects or modifies them.

```
/// <summary>
/// The Caretaker class.  This class never examines the contents of any Memento
/// and is responsible for keeping that memento.
/// </summary>
class SupplierMemory
{
    private FoodSupplierMemento _memento;

    public FoodSupplierMemento Memento
    {
        set { _memento = value; }
        get { return _memento; }
    }
}
```

Now, in our `Main()` method, we can simulate adding a new Supplier but accidentally adding the wrong address, then using the Memento to restore the old data.

```
static void Main(string[] args)
{
    //Here's a new supplier for our restaurant
    FoodSupplier s = new FoodSupplier();
    s.Name = "Harold Karstark";
    s.Phone = "(482) 555-1172";

    // Let's store that entry in our database.
    SupplierMemory m = new SupplierMemory();
    m.Memento = s.SaveMemento();

    // Continue changing originator
    s.Address = "548 S Main St. Nowhere, KS";

    // Crap, gotta undo that entry, I entered the wrong address
    s.RestoreMemento(m.Memento);

    Console.ReadKey();
```

```
}
```

Will I Ever Use This Pattern?

For websites: **Maybe? I'm really not sure.** I struggled for a long time to come with appropriate web scenarios for this that couldn't be covered by other architectures, and eventually gave up as I simply couldn't think of one. That said, I'm happy to be wrong, so if anyone would like to give real-world examples using the Memento pattern, please feel free to share in the comments!

For applications: **All the time!** The Undo function in any modern desktop app is almost certainly some form of the Memento design pattern. But since I'm primarily a web developer, I'm not aware of other users, though I'm sure they exist.

Summary

The Memento design pattern seeks to encapsulate state of an object as another object (called a Memento) and enable the ability to restore the state of the object *from* that Memento.

Oh, I almost forgot something.

```
someWriter.FinishWriting();
someWriter.RestoreState(memento); //Hungry again!
```

Day 10: Prototype

What Is This Pattern?

Prototype is a Creational design pattern in which objects are created using a prototypical instance of said object. This pattern is particularly useful for creating lots of instances of an object, all of which share some or all of their values.

The typical way of thinking about this pattern is to consider how we might model the color spectrum. There are something like 10 million visible colors, so modeling them as individual classes (e.g. `Red`, `LightMauve`, `Octarine`, `NotYellowButNotGreenEither`) would be rather impractical.

However, a color is a color, no matter what color it is; colors have the same kinds of properties as each other even if they don't have the same values for those properties. If we needed to create a lot of color instances, we could do so using the Prototype design pattern.

The Rundown

- **Type:** Creational
- **Useful?** 3/5 (Sometimes)

- **Good For:** Creating lots of similar instances.
- **Example Code:** On GitHub
- **Blog Post:** "Prototype - The Daily Design Pattern"

The Participants

- The **Prototype** declares an interface for cloning itself.
- The **ConcretePrototype** implements the cloning operation defined in the Prototype.
- The **Client** creates a new object by asking the Prototype to clone itself.

A Delicious Example

To demo this pattern, let's once again think about sandwiches. (As you can probably tell, I like sandwiches.)

In the photo to the left, there are many kinds of sandwiches. Just like the color example above, a sandwich is still a sandwich no matter what's between the two slices of bread. Let's demo how we can use the Prototype pattern to build lots of similar sandwiches.

First, we'll create an abstract class (the **Prototype** participant) to represent a sandwich, and define a method by which the abstract Sandwich class can clone itself:

```
/// <summary>
/// The Prototype abstract class
/// </summary>
abstract class SandwichPrototype
{
    public abstract SandwichPrototype Clone();
}
```

Now we need the **ConcretePrototype** participant class that can clone itself to create more Sandwich instances. For our model, we'll say that a Sandwich consists of four parts: meat, cheese, bread, and veggies.

Here's that class:

```
class Sandwich : SandwichPrototype
{
    private string Bread;
    private string Meat;
    private string Cheese;
    private string Veggies;

    public Sandwich(string bread, string meat, string cheese, string veggies)
    {
        Bread = bread;
        Meat = meat;
```

```
        Cheese = cheese;
        Veggies = veggies;
    }

    public override SandwichPrototype Clone()
    {
        string ingredientList = GetIngredientList();
        Console.WriteLine("Cloning sandwich with ingredients: {0}",
                  ingredientList.Remove(ingredientList.LastIndexOf(",")));

        return MemberwiseClone() as SandwichPrototype;
    }

    private string GetIngredientList(){ ... }
}

class SandwichMenu
{
    private Dictionary<string, SandwichPrototype> _sandwiches
      = new Dictionary<string, SandwichPrototype>();

    public SandwichPrototype this[string name]
    {
        get { return _sandwiches[name]; }
        set { _sandwiches.Add(name, value); }
    }
}
```

Now we need to create a bunch of sandwiches. In our Main() method (which does double-duty as our **Client** participant), we can do just that by instantiating the prototype and then cloning it, thereby populating our SandwichMenu object:

```
class Program
{
    static void Main(string[] args)
    {
        SandwichMenu sandwichMenu = new SandwichMenu();

        // Initialize with default sandwiches
        sandwichMenu["BLT"]
            = new Sandwich("Wheat", "Bacon", "", "Lettuce, Tomato");
        sandwichMenu["PB&J"]
            = new Sandwich("White", "", "", "Peanut Butter, Jelly");
        sandwichMenu["Turkey"]
            = new Sandwich("Rye", "Turkey", "Swiss", "Lettuce, Onion, Tomato");

        // Deli manager adds custom sandwiches
        sandwichMenu["LoadedBLT"]
            = new Sandwich("Wheat", "Turkey, Bacon", "American",
                           "Lettuce, Tomato, Onion, Olives");
        sandwichMenu["ThreeMeatCombo"]
            = new Sandwich("Rye", "Turkey, Ham, Salami",
                           "Provolone", "Lettuce, Onion");
        sandwichMenu["Vegetarian"]
```

```
            = new Sandwich("Wheat", "", "",
                           "Lettuce, Onion, Tomato, Olives, Spinach");

        // Now we can clone these sandwiches
        Sandwich sandwich1 = sandwichMenu["BLT"].Clone() as Sandwich;
        Sandwich sandwich2 = sandwichMenu["ThreeMeatCombo"].Clone() as Sandwich;
        Sandwich sandwich3 = sandwichMenu["Vegetarian"].Clone() as Sandwich;

        // Wait for user
        Console.ReadKey();
    }
}
```

By the time we get to the last line of the `Main()` method, how many total separate instances of `Sandwich` do we have? **Nine**, six in the `sandwichMenu` dictionary and three initialized as variables `sandwich1`, `sandwich2`, and `sandwich3`.

Will I Ever Use This Pattern?

Possibly. It's a good idea if you have the scenario described above. However, I'm not sure how common that scenario is in regular day-to-day coding; I haven't (consciously) implemented this pattern in several years.

The situation in which I see this pattern as being the most useful is when all of the following happens:

1. You need to create a lot of instances of an object,
2. AND those instances will be the same or similar as the prototypical instance,
3. AND creating a new instance of this object would be markedly slower than cloning an existing instance.

If you have all of those conditions, the Prototype design pattern is for you!

Summary

The Prototype pattern initializes objects by cloning them from a prototypical instance of said object. It's especially useful when you need to create many instances of related items, each of which could be slightly (but not very) different from the other instances. The primary benefit of this pattern is reduced initialization costs; by cloning many instances from a prototypical instance, you theoretically improve performance.

Day 11: Singleton

What Is This Pattern?

Singleton is a Creational design pattern in which a class is guaranteed to only ever have exactly one instance, with that instance being globally accessible.

What this means is that the pattern forces a particular object to not have an accessible constructor, and that any access performed on the object is performed upon the same instance of that object.

As you may have heard, Singleton is one of the most maligned design patterns (for reasons we will discuss in the later sections).

The Rundown

- **Type:** Creational
- **Useful?** 2/5 (Rarely, but worth knowing so you can hunt them down)
- **Good For:** Creating an object of which there can only ever be exactly one instance.
- **Example Code:** On GitHub
- **Blog Post:** "Singleton - The Daily Design Pattern"

The Participants

It's kind of silly to define participants for this design pattern, but here we go:

- The **Singleton** is a class which defines exactly one instance of itself, and that instance is globally accessible.

A Delicious Example

The theme I've been using for these examples so far is "food", but food items are not a good way to model the Singleton design pattern: there's not ever going to be a piece of food that everybody will access a single instance of (because that would be gross). Instead, let's visit our local diner and think about that little bell that sits on the counter.

In movies, one of the best ways to identify that the characters are are in a greasy diner is by having an overweight chef with a dirty apron hit a bell and yell "Order Up!". The thing about that bell is that there's probably only ever one; the sound is used to notify the servers that the next order is at the window and needs to be taken to the tables.

If there's only ever one bell, we can model that as a Singleton.

```
/// <summary>
/// Singleton
/// </summary>
public sealed class TheBell
{
    private static TheBell bellConnection;
    private static object syncRoot = new Object();
    private TheBell(){ }

    /// <summary>
    /// We implement this method to ensure thread safety for our singleton.
    /// </summary>
    public static TheBell Instance
    {
        get
        {
            lock(syncRoot)
            {
                if(bellConnection == null)
                {
                    bellConnection = new TheBell();
                }
            }

            return bellConnection;
        }
    }

    public void Ring()
    {
        Console.WriteLine("Ding! Order up!");
    }
}
```

Notice that the `TheBell` class has a private constructor. This is to ensure that it can never be instantiated, and can only be accessed through the `Instance` property.

Further, note the `syncRoot` object. This a simple object that allows our Singleton to be thread-safe; since there's only ever one, we must ensure that any thread which wants to access it has an exclusive lock on it.

This Pattern Has Problems

Singleton is probably the most maligned Design Pattern, and for good reason.

For one thing, Singletons are not global variables, though the latter is often mistaken for the former. A Singleton is a class unto itself, and global variables are just properties.

Further, many people argue that Singletons violate common guiding principles such as the Single Responsibility Principle. By its very nature, you cannot pass a Singleton to other classes, and this is often a code smell.

Mostly, though, Singletons are maligned because they are so often misused. It's entirely too easy, to paraphrase Jamie Zawinski, to see a problem, think "I know, I'll use a Singleton," and end up with two problems. Be careful that what you're using the Singleton for actually *requires* that pattern, and even then be on the lookout for a better, more appropriate manner by which you can solve your current problem.

Will I Ever Use This Pattern?

Not on purpose.

(Kidding, kidding. Sort of.)

Thing is, Singletons are (fittingly) good for one and only one purpose yet are easily understood and quick to implement, which makes them a favorite of people afflicted with golden hammer syndrome. It's all too common to find Singletons in use where global variables should be used instead.

Use the Singleton design pattern sparingly and only for its intended purpose (a single, globally accessible instance of an object) with full knowledge of this pattern's limits, and you'll find that it, like all the other design patterns, has its own set of valid uses.

Summary

Singletons are objects of which there can only ever be exactly one instance. They're not global variables and many people think they violate common principles of good software development, but they do have their uses and so should be used sparingly.

Day 12: Flyweight

What Is This Pattern?

The Flyweight design pattern is used to create lots of small, related objects without invoking a lot of overhead work in doing so, thereby improving performance and maintainability.

The idea is that each Flyweight object has two pieces:

- The **intrinsic state**, which is stored within the Flyweight object itself, and
- The **extrinsic state**, which is stored or calculated by other components.

The Flyweight design pattern allows many instances of an object to share their intrinsic state and thereby reduce the cost associated with creating them.

The Rundown

- **Type:** Structural

- **Useful?** 1/5 (Rarely)
- **Good For:** Creating lots of instances of the same set of objects and thereby improving performance.
- **Example Code:** On GitHub
- **Blog Post:** "Flyweight - The Daily Design Pattern"

The Participants

- The **Flyweight** declares an interface through which flyweights can receive and act upon extrinsic state.
- The **ConcreteFlyweight** objects implement the Flyweight interface and may be sharable. Any state stored by these objects must be intrinsic to the object.
- The **FlyweightFactory** creates and manages flyweight objects, while also ensuring that they are shared properly. When the FlyweightFactory is asked to create an object, it either uses an existing instance of that object or creates a new one if no existing one exists.
- The **Client** maintains a reference to flyweights and computes or stores the extrinsic state of said flyweights.

A Delicious Example

To model the Flyweight design pattern, let's think about sliders, as shown in the picture to the left.

For those of you who might not be familiar with the term "slider", a slider is a small hamburger, typically only 3 or 4 inches in diameter. They're often used as party snacks, but can also be a meal unto themselves.

At any rate, let's imagine that we need to create a whole bunch of these sliders for our fictional restaurant; this is a good model for Flyweight.

First, let's build a `Slider` abstract class (the **Flyweight** participant):

```
/// <summary>
/// The Flyweight class
/// </summary>
abstract class Slider
{
    protected string Name;
    protected string Cheese;
    protected string Toppings;
    protected decimal Price;

    public abstract void Display(int orderTotal);
```

```
}
```

The `Slider` class has properties for Name, Cheese, Toppings, and Price (all of which are part of the intrinsic state of these objects), and an abstract method `Display()` which will display the details of that slider.

Now we need our **ConcreteFlyweight** objects. Let's build three: one each for `BaconMaster`, `VeggieSlider`, and `BBQKing`:

```
/// <summary>
/// A  ConcreteFlyweight class
/// </summary>
class BaconMaster : Slider
{
    public BaconMaster()
    {
        Name = "Bacon Master";
        Cheese = "American";
        Toppings = "lots of bacon";
        Price = 2.39m;
    }

    public override void Display(int orderTotal)
    {
        Console.WriteLine("Slider #" + orderTotal + ": "
                                     + Name + " - topped with "
                                     + Cheese + " cheese and "
                                     + Toppings + "! $" + Price.ToString());
    }
}

/// <summary>
/// A ConcreteFlyweight class
/// </summary>
class VeggieSlider : Slider
{
    public VeggieSlider()
    {
        Name = "Veggie Slider";
        Cheese = "Swiss";
        Toppings = "lettuce, onion, tomato, and pickles";
        Price = 1.99m;
    }

    public override void Display(int orderTotal)
    {
        Console.WriteLine("Slider #" + orderTotal + ": "
                                + Name + " - topped with "
                                + Cheese + " cheese and "
                                + Toppings + "! $"
                                + Price.ToString());
    }
```

```
}

/// <summary>
/// A ConcreteFlyweight class
/// </summary>
class BBQKing : Slider
{
    public BBQKing()
    {
        Name = "BBQ King";
        Cheese = "American";
        Toppings = "Onion rings, lettuce, and BBQ sauce";
        Price = 2.49m;
    }

    public override void Display(int orderTotal)
    {
        Console.WriteLine("Slider #" + orderTotal
                                    + ": " + Name + " - topped with "
                                    + Cheese + " cheese and "
                                    + Toppings + "! $"
                                    + Price.ToString());
    }
}
```

Note that the ConcreteFlyweight classes are, of course, very similar to one another: they all have the same properties. This is critical to using Flyweight: all of the related objects must have the same definition (or at least reasonably close to the same definition).

Finally, we need our **FlyweightFactory** participant, which will create Flyweight objects. The Factory will store a collection of already-created sliders, and any time another slider of the same type needs to be created, the Factory will use the already-created one rather than creating a brand-new one.

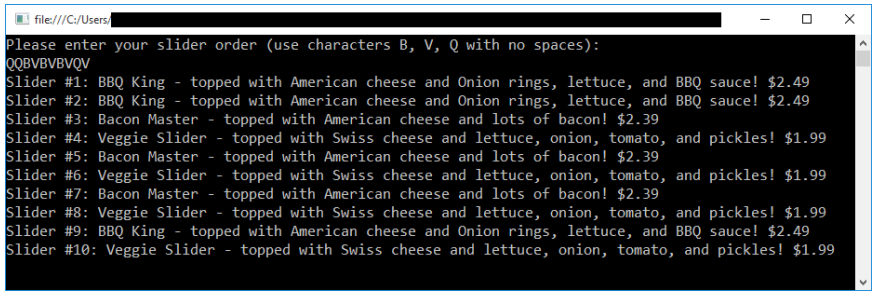
```
/// <summary>
/// The FlyweightFactory class
/// </summary>
class SliderFactory
{
    private Dictionary<char, Slider> _sliders =
        new Dictionary<char, Slider>();

    public Slider GetSlider(char key)
    {
        Slider slider = null;
        if (_sliders.ContainsKey(key)) //If we've already created one of the
requested type of slider, just use that.
        {
            slider = _sliders[key];
        }
        else //Otherwise, create a brand new instance of the slider.
        {
            switch (key)
            {
```

```
                case 'B': slider = new BaconMaster(); break;
                case 'V': slider = new VeggieSlider(); break;
                case 'Q': slider = new BBQKing(); break;
            }
            _sliders.Add(key, slider);
        }
        return slider;
    }
}
```

All of this comes together in our Main() method (which is also our **Client** participant). Let's pretend we are an order system and we need to take orders for these sliders; the patron can order as many kinds of sliders as s/he wants. Given those parameters, the implementation of our Main() method might look something like the following example:

```
static void Main(string[] args)
{
    // Build a slider order using patron's input
    Console.WriteLine(
      "Please enter your slider order (use characters B, V, Z with no spaces):"
    );
    var order = Console.ReadLine();
    char[] chars = order.ToCharArray();

    SliderFactory factory = new SliderFactory();

    int orderTotal = 0;

    //Get the slider from the factory
    foreach (char c in chars)
    {
        orderTotal++;
        Slider character = factory.GetSlider(c);
        character.Display(orderTotal);
    }

    Console.ReadKey();
}
```

The output from this sample app might look something like the screenshot to the right.



Will I Ever Use This Pattern?

Probably not. In theory, this pattern could improve performance, but in practice it's limited to scenarios where you find yourself creating a lot of objects from one or more templates. Further, the entire point of this pattern is to improve performance, and in my opinion performance is not an issue until you can prove that it is, so while refactoring to this pattern may be useful in some extreme

circumstances, for most people and most projects the overhead and complexity of the Flyweight pattern outweigh the benefits.

In my opinion, if you need to create lots of instances of an object, you'd be better off using something like the Prototype design pattern rather than Flyweight.

Summary

The Flyweight pattern strives to improve performance by creating lots of objects from a small set of "template" objects, where those objects are the same or very similar to all their other instances. In practice, though, the usefulness of this pattern is limited, and you might be better off using Prototype. That said, if anyone has a different opinion on the benefits of Flyweight, I'd love to hear about it, so share in the comments!

Day 13: Builder

What Is This Pattern?

The Builder pattern separates the construction of an object from its representation so that the same construction process can create different representations.

The general idea is that the order in which things happen when an object is instantiated will be the same, but the actual details of those steps change based upon what the concrete implementation is.

The Rundown

- **Type:** Creational
- **Useful?** 1/5 (Probably not)
- **Good For:** Creating objects which need several steps to happen in order, but the steps are different for different specific implementations.
- **Example Code:** On GitHub
- **Blog Post:** "Builder - The Daily Design Pattern"

The Participants

- The **Builder** specifies an abstract interface for creating parts of a Product.
- The **ConcreteBuilder** constructs and assembles parts of the product by implementing the Builder interface. It must also define and track the representation it creates.
- The **Product** represents the object being constructed. It includes classes for defining the parts of the object, including any interfaces for assembling the parts into the final result.
- The **Director** constructs an object using the Builder interface.

A Delicious Example

To demonstrate how the Builder design pattern works, we once again turn our hungry eyes to that most portable and simple of lunch foods: the humble sandwich.

Here's the thing about sandwiches: the only thing that defines a sandwich is something edible between two slices of bread. That's it. This means that (as discussed in Day 1: Factory Method), a hot dog is a sandwich, which seems like a ridiculous statement but is technically correct.

That said, different types of sandwiches require different steps in order to make them, but they're still just sandwiches. Most of the time, the same kinds of ingredients will be used to create many different kinds of sandwiches. Let's see how we can use the Builder pattern to build us some of these yummy sandwiches.

To start off, we need to implement the **Director** participant. We'll call our Director `AssemblyLine`, make it a class, and it will define in what steps the process of making a sandwich are called.

```
/// <summary>
/// The Director
/// </summary>
class AssemblyLine
{
    // Builder uses a complex series of steps
    //
    public void Assemble(SandwichBuilder sandwichBuilder)
    {
        sandwichBuilder.AddBread();
        sandwichBuilder.AddMeats();
        sandwichBuilder.AddCheese();
        sandwichBuilder.AddVeggies();
        sandwichBuilder.AddCondiments();
    }
}
```

We also need to define the **Product** participant which is being built by the Builder participant. For this demo, the Product is, of course, a `Sandwich`.

```
/// <summary>
/// The Product class
/// </summary>
class Sandwich
{
    private string _sandwichType;
    private Dictionary<string, string> _ingredients = new Dictionary<string,
string>();
```

```
    // Constructor
    public Sandwich(string sandwichType)
    {
        this._sandwichType = sandwichType;
    }

    // Indexer
    public string this[string key]
    {
        get { return _ingredients[key]; }
        set { _ingredients[key] = value; }
    }

    public void Show()
    {
        Console.WriteLine("\n---------------------------");
        Console.WriteLine("Sandwich: {0}", _sandwichType);
        Console.WriteLine(" Bread: {0}", _ingredients["bread"]);
        Console.WriteLine(" Meat: {0}", _ingredients["meat"]);
        Console.WriteLine(" Cheese: {0}", _ingredients["cheese"]);
        Console.WriteLine(" Veggies: {0}", _ingredients["veggies"]);
        Console.WriteLine(" Condiments: {0}", _ingredients["condiments"]);
    }
}
```

Now that we know the definition of the product we are building, let's now create the **Builder** participant - an abstract class **SandwichBuilder**:

```
/// <summary>
/// The Builder abstract class
/// </summary>
abstract class SandwichBuilder
{
    protected Sandwich sandwich;

    // Gets sandwich instance
    public Sandwich Sandwich
    {
        get { return sandwich; }
    }

    // Abstract build methods
    public abstract void AddBread();
    public abstract void AddMeats();
    public abstract void AddCheese();
    public abstract void AddVeggies();
    public abstract void AddCondiments();
}
```

Notice the five abstract methods. Each subclass of `SandwichBuilder` will need to implement those methods in order to properly build a sandwich.

Next, let's implement a few **ConcreteBuilder** classes to build some specific sandwiches.

```
/// <summary>
/// A ConcreteBuilder class
/// </summary>
class TurkeyClub : SandwichBuilder
{
    public TurkeyClub()
    {
        sandwich = new Sandwich("Turkey Club");
    }

    public override void AddBread()
    {
        sandwich["bread"] = "12-Grain";
    }

    public override void AddMeats()
    {
        sandwich["meat"] = "Turkey";
    }

    public override void AddCheese()
    {
        sandwich["cheese"] = "Swiss";
    }

    public override void AddVeggies()
    {
        sandwich["veggies"] = "Lettuce, Tomato";
    }

    public override void AddCondiments()
    {
        sandwich["condiments"] = "Mayo";
    }
}


/// <summary>
/// A ConcreteBuilder class
/// </summary>
class BLT : SandwichBuilder
{
    public BLT()
    {
        sandwich = new Sandwich("BLT");
    }

    public override void AddBread()
    {
        sandwich["bread"] = "Wheat";
    }

    public override void AddMeats()
```

```
    {
        sandwich["meat"] = "Bacon";
    }

    public override void AddCheese()
    {
        sandwich["cheese"] = "None";
    }

    public override void AddVeggies()
    {
        sandwich["veggies"] = "Lettuce, Tomato";
    }

    public override void AddCondiments()
    {
        sandwich["condiments"] = "Mayo, Mustard";
    }
}
/// <summary>
/// A ConcreteBuilder class
/// </summary>
class HamAndCheese : SandwichBuilder
{
    public HamAndCheese()
    {
        sandwich = new Sandwich("Ham and Cheese");
    }

    public override void AddBread()
    {
        sandwich["bread"] = "White";
    }

    public override void AddMeats()
    {
        sandwich["meat"] = "Ham";
    }

    public override void AddCheese()
    {
        sandwich["cheese"] = "American";
    }

    public override void AddVeggies()
    {
        sandwich["veggies"] = "None";
    }

    public override void AddCondiments()
    {
        sandwich["condiments"] = "Mayo";
    }
}
```

Once we have all the ConcreteBuilder classes written up, we can use them in our `Main()` method like so:

```
static void Main(string[] args)
{
    SandwichBuilder builder;

    // Create shop with sandwich assembly line
    AssemblyLine shop = new AssemblyLine();

    // Construct and display sandwiches
    builder = new HamAndCheese();
    shop.Assemble(builder);
    builder.Sandwich.Show();

    builder = new BLT();
    shop.Assemble(builder);
    builder.Sandwich.Show();

    builder = new TurkeyClub();
    shop.Assemble(builder);
    builder.Sandwich.Show();

    // Wait for user
    Console.ReadKey();
}
```

The nice thing about this pattern is that we can now reuse the `AssemblyLine` class on any `SandwichBuilder` we wish, and we have more fine-grained control over how the sandwiches are built.

(Yes, that pun was intentional. No, I am not sorry.)

Will I Ever Use This Pattern?

Probably not. Let's face it, this is a lot of work to build these supposedly related items in a reusable manner. The patterns some degree of assumptions about how these objects should be created, and for me it's too many assumptions to rely on using this pattern in common projects. Seems to me like the Builder pattern has some uses, just not a lot of them.

Summary

The Builder pattern allows us to build related sets of objects with the same steps, but leaving the implementation of those steps up to the subclasses. However, it's a pain to set up, and overall it's not terribly useful as far as I can see.

Day 14: State

What Is This Pattern?

The State design pattern seeks to allow an object to change its own behavior when its internal state changes.

In this pattern, the "states" in which an object can exist are classes unto themselves, which refer back to the object instance and cause that instance's behaviors to differ depending on the state it is currently residing in.

The Rundown

- **Type:** Behavioral
- **Useful?** 3/5 (Sometimes)
- **Good For:** Allowing an object's behavior to change as its internal state does.
- **Example Code:** On GitHub
- **Blog Post:** "State - The Daily Design Pattern"

The Participants

- The **Context** defines an interface of interest to the clients. It also maintains a reference to an instance of ConcreteState which represents the current state.
- The **State** defines an interface for encapsulating the behavior of the object associated with a particular state.
- The **ConcreteState** objects are subclasses which each implement a behavior (or set of behaviors) associated with a state of the Context.

A Delicious Example

Those of you who don't like red meat, turn back now. For everyone else, let's talk about steaks; specifically, how to cook them to different temperatures.

The United States Food and Drug Administration sets guidelines as to how thoroughly cooked a steak must be in order to be a) safe to eat, and b) considered a certain level of "doneness" (which is rapidly becoming my new favorite word).

For many steaks, the levels of doneness are:

- Uncooked
- Rare
- Medium Rare

- Medium
- Medium Well
- Well Done

Let's implement a system which keeps track of the internal temperature of a steak and assigns a level of doneness to it. We can model this using the State design pattern.

First, let's define our **State** participant, which represents a "doneness" level for a steak (and maintains a reference to an actual `Steak` instance).

```
/// <summary>
/// The State abstract class
/// </summary>
abstract class Doneness
{
    protected Steak steak;
    protected double currentTemp;
    protected double lowerTemp;
    protected double upperTemp;
    protected bool canEat;

    public Steak Steak
    {
        get { return steak; }
        set { steak = value; }
    }

    public double CurrentTemp
    {
        get { return currentTemp; }
        set { currentTemp = value; }
    }

    public abstract void AddTemp(double temp);
    public abstract void RemoveTemp(double temp);
    public abstract void DonenessCheck();
}
```

The abstract methods `AddTemp()`, `RemoveTemp()`, and `DonenessCheck()` will need to be implemented by each of the states we can place the steak in.

Now that we have the State participant, let's define some **ConcreteState** objects. First, let's define a state for when the steak is uncooked and, therefore, not safe to eat. In this state, we can add cook temperature and remove cook temperature, but the steak will not be safe to eat until the cook temp is above 130 degrees Fahrenheit (54.4 degrees Celsius).

We also need to implement the method `DonenessCheck()`, which determines whether or not the internal temperature of the steak is sufficiently high enough to allow it to move to another state. In this case, we'll make the assumption that a steak may only move to the next state of

`Rare`.

```
/// <summary>
/// A Concrete State class.
/// </summary>
class Uncooked : Doneness
{
    public Uncooked(Doneness state)
    {
        currentTemp = state.CurrentTemp;
        steak = state.Steak;
        Initialize();
    }

    private void Initialize()
    {
        lowerTemp = 0;
        upperTemp = 130;
        canEat = false;
    }

    public override void AddTemp(double amount)
    {
        currentTemp += amount;
        DonenessCheck();
    }

    public override void RemoveTemp(double amount)
    {
        currentTemp -= amount;
        DonenessCheck();
    }

    public override void DonenessCheck()
    {
        if (currentTemp > upperTemp)
        {
            steak.State = new Rare(this);
        }
    }
}
```

Now let's think about the first edible state, `Rare`. In this state, we can add and remove cook temperature, and the steak is now edible (so we must initialize it as such).

```
/// <summary>
/// A 'ConcreteState' class.
/// </summary>
class Rare : Doneness
{
    public Rare(Doneness state) : this(state.CurrentTemp, state.Steak){}

    public Rare(double currentTemp, Steak steak)
```

```
    {
        this.currentTemp = currentTemp;
        this.steak = steak;
        canEat = true; //We can now eat the steak
        Initialize();
    }

    private void Initialize()
    {
        lowerTemp = 130;
        upperTemp = 139.999999999999;
        canEat = true;
    }

    public override void AddTemp(double amount)
    {
        currentTemp += amount;
        DonenessCheck();
    }

    public override void RemoveTemp(double amount)
    {
        currentTemp -= amount;
        DonenessCheck();
    }

    public override void DonenessCheck()
    {
        if (currentTemp < lowerTemp)
        {
            steak.State = new Uncooked(this);
        }
        else if (currentTemp > upperTemp)
        {
            steak.State = new MediumRare(this);
        }
    }
}
```

In a similar vein, we can implement the states for `MediumRare`, `Medium`, and `WellDone`.

```
/// <summary>
/// A Concrete State class
/// </summary>
class MediumRare : Doneness
{
    public MediumRare(Doneness state) : this(state.CurrentTemp, state.Steak)
    {
    }

    public MediumRare(double currentTemp, Steak steak)
    {
        this.currentTemp = currentTemp;
```

```
        this.steak = steak;
        canEat = true;
        Initialize();
    }

    private void Initialize()
    {
        lowerTemp = 140;
        upperTemp = 154.9999999999;
    }

    public override void AddTemp(double amount)
    {
        currentTemp += amount;
        DonenessCheck();
    }

    public override void RemoveTemp(double amount)
    {
        currentTemp -= amount;
        DonenessCheck();
    }

    public override void DonenessCheck()
    {
        if (currentTemp < 0.0)
        {
            steak.State = new Uncooked(this);
        }
        else if (currentTemp < lowerTemp)
        {
            steak.State = new Rare(this);
        }
        else if (currentTemp > upperTemp)
        {
            steak.State = new Medium(this);
        }
    }
}
/// <summary>
/// A Concrete State class
/// </summary>
class Medium : Doneness
{
    public Medium(Doneness state) : this(state.CurrentTemp, state.Steak)
    {
    }

    public Medium(double currentTemp, Steak steak)
    {
        this.currentTemp = currentTemp;
        this.steak = steak;
        canEat = true;
        Initialize();
```

```
    }

    private void Initialize()
    {
        lowerTemp = 155;
        upperTemp = 169.9999999999;
    }

    public override void AddTemp(double amount)
    {
        currentTemp += amount;
        DonenessCheck();
    }

    public override void RemoveTemp(double amount)
    {
        currentTemp -= amount;
        DonenessCheck();
    }

    public override void DonenessCheck()
    {
        if (currentTemp < 130)
        {
            steak.State = new Uncooked(this);
        }
        else if (currentTemp < lowerTemp)
        {
            steak.State = new MediumRare(this);
        }
        else if (currentTemp > upperTemp)
        {
            steak.State = new WellDone(this);
        }
    }
}

/// <summary>
/// A Concrete State class
/// </summary>
class WellDone : Doneness //aka Ruined
{
    public WellDone(Doneness state) : this(state.CurrentTemp, state.Steak)
    {
    }

    public WellDone(double currentTemp, Steak steak)
    {
        this.currentTemp = currentTemp;
        this.steak = steak;
        canEat = true;
        Initialize();
    }
```

```
    private void Initialize()
    {
        lowerTemp = 170;
        upperTemp = 230;
    }

    public override void AddTemp(double amount)
    {
        currentTemp += amount;
        DonenessCheck();
    }

    public override void RemoveTemp(double amount)
    {
        currentTemp -= amount;
        DonenessCheck();
    }

    public override void DonenessCheck()
    {
        if (currentTemp < 0)
        {
            steak.State = new Uncooked(this);
        }
        else if (currentTemp < lowerTemp)
        {
            steak.State = new Medium(this);
        }
    }
}
```

Now that we have all of our states defined, we can finally implement our **Context** participant. In this case, the Context is a `Steak` class which maintains a reference to the `Doneness` state it is currently in. Further, whenever we add or remove temperature from the steak, it must call the current `Doneness` state's corresponding method.

```
/// <summary>
/// The Context class
/// </summary>
class Steak
{
    private Doneness _state;
    private string _beefCut;

    public Steak(string beefCut)
    {
        _cook = beefCut;
        _state = new Rare(0.0, this);
    }

    public double CurrentTemp
    {
```

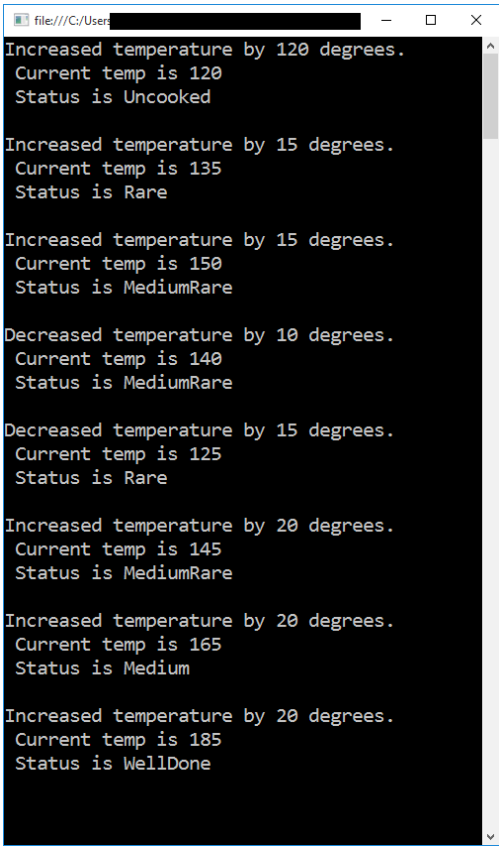
```
        get { return _state.CurrentTemp; }
    }

    public Doneness State
    {
        get { return _state; }
        set { _state = value; }
    }

    public void AddTemp(double amount)
    {
        _state.AddTemp(amount);
        Console.WriteLine("Increased temperature by {0} degrees.", amount);
        Console.WriteLine(" Current temp is {0}", CurrentTemp);
        Console.WriteLine(" Status is {0}", State.GetType().Name);
        Console.WriteLine("");
    }

    public void RemoveTemp(double amount)
    {
        _state.RemoveTemp(amount);
        Console.WriteLine("Decreased temperature by {0} degrees.", amount);
        Console.WriteLine(" Current temp is {0}", CurrentTemp);
        Console.WriteLine(" Status is {0}", State.GetType().Name);
        Console.WriteLine("");
    }
}
```

In our `Main()` method, we can use these states by creating a `Steak` object and then changing its internal temperature

```
static void Main(string[] args)
{
    //Let's cook a steak!
    Steak account = new Steak("T-Bone");

    // Apply temperature changes
    account.AddTemp(120);
    account.AddTemp(15);
    account.AddTemp(15);
    account.RemoveTemp(10); //Yes I know cooking doesn't work this way
    account.RemoveTemp(15);
    account.AddTemp(20);
    account.AddTemp(20);
    account.AddTemp(20);

    Console.ReadKey();
}
```

As we change the temperature, we change the state of the `Steak` object. When the `Steak` class's internal temperature changes, the `Doneness` state in which it currently resides also

changes, and consequently the apparent behavior of that object shifts to whatever behavior is defined by the current state.



Will I Ever Use This Pattern?

Sometimes, more often if you deal with objects which change behaviors as their internal state changes. I personally have a lot of experience with this pattern, as I built a Workflow Engine database which is this pattern writ large and made changeable. If you'd like an example of how the State design pattern would be used in the real world, go check out that series of posts.

Summary

The State pattern allows the behavior of an object to change as its internal state changes, and it accomplishes this by making the states of an object separate classes from the object itself. Consequently, the states can implement their own behavior for the object, and the object can "react" to its internal state changing.

Day 15: Strategy

What Is This Pattern?

The Strategy design pattern defines a family of algorithms, then makes them interchangeable by encapsulating each as an object. Consequently, the actual operation of the algorithm can vary based on other inputs, such as which client is using it.

The basic idea of this pattern is that if we encapsulate behavior as objects, we can then select which object to use and, thereby, which behavior to implement based upon some external inputs or state. We further allow for many different behaviors to be implemented without creating huge if/then or switch statements.

The Rundown

- **Type:** Behavioral
- **Useful?** 4/5 (Very)
- **Good For:** Encapsulating parts of an algorithm as objects and allowing them to be invoked independently.
- **Example Code:** On GitHub
- **Blog Post:** "Strategy - The Daily Design Pattern"

The Participants

- The **Strategy** declares an interface which is implemented by all supported algorithms.
- The **ConcreteStrategy** objects implement the algorithm defined by the Strategy.
- The **Context** maintains a reference to a Strategy object, and uses that reference to call the algorithm defined by a particular ConcreteStrategy.

A Delicious Example

To model this pattern, let's talk about some different ways to cook food.

When cooking various kinds of food, particularly meats, there's often more than one way to cook them to safe eating temperatures. For example, you might grill them, bake them, deep-fry them, or broil them, depending on whether you have friends over, how much you want to show off your cooking skills, and how many burns you are willing to suffer. Each of these methods will get the item cooked, just via different processes. These processes, in object-oriented code using the Strategy pattern, can each be their own class.

In our example, let's pretend that we'll ask the user what method they'd like to use to cook their food, and then implement that method using the Strategy design pattern.

First, let's write up the Strategy participant, which for our demo is an abstract class.

```
/// <summary>
/// The Strategy abstract class, which defines an interface common to all
supported strategy algorithms.
/// </summary>
abstract class CookStrategy
{
    public abstract void Cook(string food);
}
```

With this setup, each strategy by which we will cook a food item must implement the method `Cook()`. Let's implement a few of those strategies now (these are all **ConcreteStrategy** participants):

```
/// <summary>
/// A Concrete Strategy class
/// </summary>
class Grilling : CookStrategy
{
    public override void Cook(string food)
```

```
    {
        Console.WriteLine("\nCooking " + food + " by grilling it.");
    }
}

/// <summary>
/// A Concrete Strategy class
/// </summary>
class OvenBaking : CookStrategy
{
    public override void Cook(string food)
    {
        Console.WriteLine("\nCooking " + food + " by oven baking it.");
    }
}

/// <summary>
/// A Concrete Strategy class
/// </summary>
class DeepFrying : CookStrategy
{
    public override void Cook(string food)
    {
        Console.WriteLine("\nCooking " + food + " by deep frying it");
    }
}
```

The only thing left to do to complete demo app is to implement our **Context** participant. Remember that the Context maintains a reference to both the food we are cooking and the Strategy we are using to do so. Fittingly, we'll call our Context class `CookingMethod`.

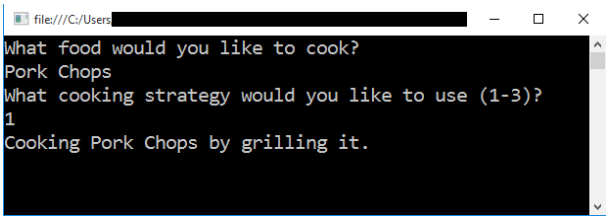
```
/// <summary>
/// The Context class, which maintains a reference to the chosen Strategy.
/// </summary>
class CookingMethod
{
    private string Food;
    private CookStrategy _cookStrategy;

    public void SetCookStrategy(CookStrategy cookStrategy)
    {
        this._cookStrategy = cookStrategy;
    }

    public void SetFood(string name)
    {
        Food = name;
    }

    public void Cook()
    {
        _cookStrategy.Cook(Food);
        Console.WriteLine();
    }
```

```
}
```

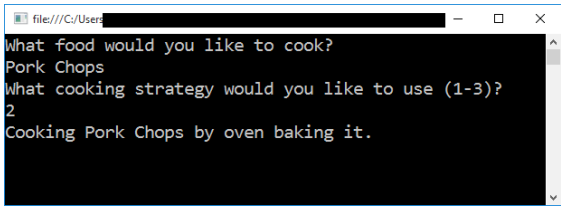
To wrap it all up, we can allow the user to select what food they want to cook and what Strategy they wish to use in our `Main()` method.

```
static void Main(string[] args)
{
    CookingMethod cookMethod = new CookingMethod();

    Console.WriteLine("What food would you like to cook?");
    var food = Console.ReadLine();
    cookMethod.SetFood(food);

    Console.WriteLine("What cooking strategy would you like to use (1-3)?");
    int input = int.Parse(Console.ReadKey().KeyChar.ToString());

    switch(input)
    {
        case 1:
            cookMethod.SetCookStrategy(new Grilling());
            cookMethod.Cook();
            break;

        case 2:
            cookMethod.SetCookStrategy(new OvenBaking());
            cookMethod.Cook();
            break;

        case 3:
            cookMethod.SetCookStrategy(new DeepFrying());
            cookMethod.Cook();
            break;

        default:
            Console.WriteLine("Invalid Selection!");
            break;
    }
    Console.ReadKey();
}
```

In the screenshots on this page and the next, you can see three examples of output from the sample application we've built here.



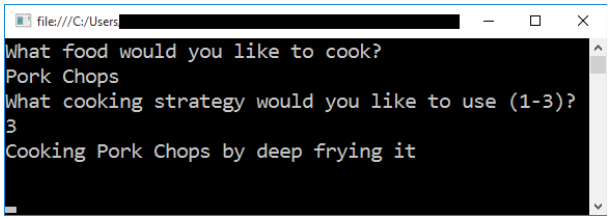
Will I Ever Use This Pattern?



Probably. I find this pattern to be very useful when refactoring applications which have many different rules regarding how objects behave, particularly in our line-of-business apps which often have many different possible strategies in play. If you're only

ever going to have two or three strategies for a given object, refactoring to the Strategy pattern may not be worth it, but if you could possibly have more, that's where this design pattern shines.

Strategy vs State



It's worthwhile to note that the Strategy pattern and the State pattern are similar, but are used in different ways.

- The Strategy pattern decides on an appropriate behavior based on external (relative to the object) inputs, whereas the State pattern decides on an appropriate behavior based on the object's internal state.
- Objects in the State pattern store a reference to the object that is in that state; no such thing occurs when using Strategy.
- Strategies (generally) handle only a single, specific task, whereas States can be as complex as necessary to properly represent the desired state of an object.

Summary

The Strategy design pattern allows different behaviors for a given object to be used under different circumstances. This allows for many different behaviors to be implemented and tested separately, since each will be encapsulated as an object.

Day 16: Proxy

What Is This Pattern?

The Proxy pattern provides a surrogate or placeholder object to control access to another, different object. The Proxy object can be used in the same manner as its containing object.

The Proxy object can then hide or change data on the hidden object, or otherwise manipulate its behavior. However, the Proxy must still be able to be used anywhere the hidden object is.

The Rundown

- **Type:** Structural
- **Useful?** 4/5 (Very)
- **Good For:** Controlling access to a particular object, testing scenarios.
- **Example Code:** On GitHub
- **Blog Post:** "Proxy - The Daily Design Pattern"

The Participants

- The **Subject** defines a common interface for the RealSubject and the Proxy such that the Proxy can be used anywhere the RealSubject is expected.
- The **RealSubject** defines the concrete object which the Proxy represents.
- The **Proxy** maintains a reference to the RealSubject and controls access to it. It must implement the same interface as the RealSubject so that the two can be used interchangeably.

A Delicious Example

To demonstrate how to use the Proxy design pattern in real-world code, let's talk about servers in a high-end restaurant, as we did for Day 3: Façade and Day 4: Adapter.

For this day's demo, let's imagine that servers at a restaurant primarily do three things:

1. Take the patron's order.
2. Deliver the patron's order
3. Process the patron's payment

With these assumptions, we can create an interface for these actions (this interface being the **Subject** participant):

```
/// <summary>
/// The Subject interface which both the RealSubject and proxy will need to
implement
/// </summary>
public interface IServer
{
    void TakeOrder(string order);
    string DeliverOrder();
    void ProcessPayment(string payment);
}
```

Now let's create a real `Server` class (the **RealSubject** participant), which must implement the `IServer` interface:

```
/// <summary>
/// The RealSubject class which the Proxy can stand in for
/// </summary>
class Server : IServer
{
    private string Order;
    public void TakeOrder(string order)
    {
        Console.WriteLine("Server takes order for " + order + ".");
```

```
        Order = order;
    }

    public string DeliverOrder()
    {
        return Order;
    }

    public void ProcessPayment(string payment)
    {
        Console.WriteLine("Payment for order (" + payment + ") processed.");
    }
}
```

With the `IServer` interface and the `Server` class, we can now represent a server at our restaurant.

Imagine for a second that our `Server` instance is an experienced server who is helping train a newly-employed server. That new employee, from the patron's perspective, is still a server and will still behave as such. However, the new trainee cannot process payments yet, as he must first learn the ropes of taking and delivering orders.

We can create a **Proxy** object to model this new trainee. The Proxy will need to maintain a reference back to the `Server` instance so that it can call that instance's `ProcessPayment()` method:

```
/// <summary>
/// The Proxy class, which can substitute for the Real Subject.
/// </summary>
class NewServerProxy : IServer
{
    private string Order;
    private Server _server = new Server();

    public void TakeOrder(string order)
    {
        Console.WriteLine("New trainee server takes order for " + order + ".");
        Order = order;
    }

    public string DeliverOrder()
    {
        return Order;
    }

    public void ProcessPayment(string payment)
    {
        Console.WriteLine("New trainee cannot process payments yet!")
        _server.ProcessPayment(payment);
    }
}
```

As you can see, the `NewServerProxy` implements its own `TakeOrder()` and `DeliverOrder()`

methods, and calls the `Server` class's `ProcessPayment()` method. Since they both implement `IServer`, an instance of `NewServerProxy` can be used any place an instance of `Server` can be used.

Will I Ever Use This Pattern?

Probably. If you've ever had a need to change the behavior of an existing object without actually changing the definition of that object, the Proxy pattern can allow you to do that. Further, I can see this being very useful in testing scenarios, where you might need to replicate a class's behavior without fully implementing it.

Summary

The Proxy pattern seeks to create a "stand-in" object which can be used in place of an existing object and maintains a reference to an instance of said existing object. To fulfill the pattern, the Proxy object must be able to be used anywhere the replaced object can be used.

And, while you're here, check out the wine specials. We've got a lot of great options, so there's sure to be something to suit your taste.

Day 17: Decorator

What Is This Pattern?

The Decorator design pattern seeks to add new functionality to an existing object without changing that object's definition.

In other words, it wants to add new responsibilities to an individual instance of an object, without adding those responsibilities to the *class* of objects. Decorator can be thought of as an alternative to inheritance, one where instances rather than classes inherit behaviors and properties.

The Rundown

- **Type:** Structural
- **Useful?** 3/5 (Sometimes)
- **Good For:** Injecting new functionality into instances of objects at runtime rather than including that functionality in the class of objects.
- **Example Code:** On GitHub
- **Blog Post:** "Decorator - The Daily Design Pattern"

The Participants

- The **Component** defines the interface for objects which will have responsibilities or abilities added to them dynamically.
- The **ConcreteComponent** objects are objects to which said responsibilities are added.
- The **Decorator** maintains a reference to a Component and defines and interface that conforms to the Component interface.
- The ConcreteDecorator objects are the classes which actually add responsibilities to the ConcreteComponent classes.

A Delicious Example

Continuing with the restaurant theme from Day 16: Proxy, to demonstrate the Decorator design pattern, let's pretend that we run a farm-to-table restaurant.

The idea of our restaurant is that we only make dishes from ingredients that are available from our farm; that is, we can only make meals from the crops that we grow. Further, sometimes we get a rush on certain dishes, and when this happens we occasionally need to stop selling particular dishes until we can harvest more ingredients (after all, veggies don't grow overnight). In this case, then, we need to be able to mark certain dishes as "sold out" once we run out of ingredients.

To model this, let's first define our **Component** participant, which is our abstract `RestaurantDish` class:

```
/// <summary>
/// The abstract Component class
/// </summary>
abstract class RestaurantDish
{
    public abstract void Display();
}
```

As with all the other demos, where there's an abstract class or interface, there's sure to be a concrete implementation of it. In this case, we need a couple **ConcreteComponent** participant classes representing the individual dishes we serve. These classes only implement their properties, not the number of dishes available (which is the responsibility of the Decorator).

```
/// <summary>
/// A ConcreteComponent class
/// </summary>
```

```
class FreshSalad : RestaurantDish
{
    private string _greens;
    private string _cheese; //I am going to use this pun everywhere I can
    private string _dressing;

    public FreshSalad(string greens, string cheese, string dressing)
    {
        _greens = greens;
        _cheese = cheese;
        _dressing = dressing;
    }

    public override void Display()
    {
        Console.WriteLine("\nFresh Salad:");
        Console.WriteLine(" Greens: {0}", _greens);
        Console.WriteLine(" Cheese: {0}", _cheese);
        Console.WriteLine(" Dressing: {0}", _dressing);
    }
}

/// <summary>
/// A ConcreteComponent class
/// </summary>
class Pasta : RestaurantDish
{
    private string _pastaType;
    private string _sauce;

    public Pasta(string pastaType, string sauce)
    {
        _pastaType = pastaType;
        _sauce = sauce;
    }

    public override void Display()
    {
        Console.WriteLine("\nClassic Pasta:");
        Console.WriteLine(" Pasta: {0}", _pastaType);
        Console.WriteLine(" Sauce: {0}", _sauce);
    }
}
```

Now that we've got our ConcreteComponents defined, we will need to decorate those dishes at runtime with the ability to keep track of whether or not we've exhausted all the ingredients. To accomplish this, let's first implement a `Decorator` abstract class (which, in a rare case of name matching purpose, is also our **Decorator** participant):

```
/// <summary>
/// The abstract Decorator class.
/// </summary>
abstract class Decorator : RestaurantDish
{
```

```
    protected RestaurantDish _dish;

    public Decorator(RestaurantDish dish)
    {
        _dish = dish;
    }

    public override void Display()
    {
        _dish.Display();
    }
}
```

Finally, we need a **ConcreteDecorator** for keeping track of how many of the dishes have been ordered. Of course, the concrete decorator must inherit from `Decorator`.

```
/// <summary>
/// A ConcreteDecorator. This class will impart "responsibilities"
/// onto the dishes(e.g. whether or not those dishes have
/// enough ingredients left to order them)
/// </summary>
class Available : Decorator
{
    public int NumAvailable { get; set; } //How many can we make?
    protected List<string> customers = new List<string>();
    public Available(RestaurantDish dish, int numAvailable) : base(dish)
    {
            NumAvailable = numAvailable;
    }

    public void OrderItem(string name)
    {
        if (NumAvailable > 0)
        {
            customers.Add(name);
            NumAvailable--;
        }
        else
        {
            Console.WriteLine("\nNot enough ingredients for "
                                + name + "'s order!");
        }
    }

    public override void Display()
    {
        base.Display();

        foreach(var customer in customers)
        {
            Console.WriteLine("Ordered by " + customer);
        }
    }
}
```

Once again, we arrive at the point where we can make the demo come together in the `Main()` method. Here's how we'll make that happen:

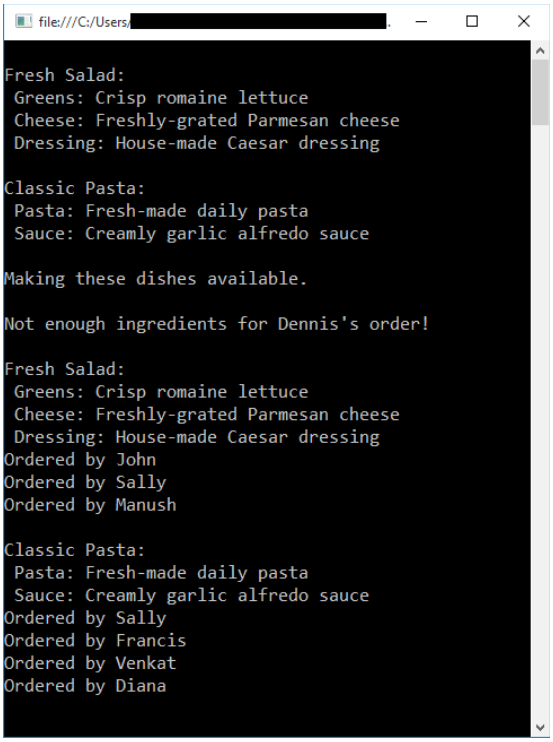
- First, we define a set of dishes that our restaurant can make
- Next, we decorate those dishes so that when we run out of ingredients we can notify the affected patrons.
- Finally, we have some patrons order the created dishes.

When all that gets put together, it looks like this:

```
static void Main(string[] args)
{
    //Step 1: Define some dishes, and how many of each we can make
    FreshSalad caesarSalad = new FreshSalad("Crisp romaine lettuce",
                                            "Freshly-grated Parmesan cheese",
                                            "House-made Caesar dressing");
    caesarSalad.Display();

    Pasta fettuccineAlfredo = new Pasta("Fresh-made daily pasta",
                                        "Creamly garlic alfredo sauce");
    fettuccineAlfredo.Display();

    Console.WriteLine("\nMaking these dishes available.");

    //Step 2: Decorate the dishes; now if we attempt to order them once we're
    // out of ingredients, we can notify the customer
    Available caesarAvailable = new Available(caesarSalad, 3);
    Available alfredoAvailable = new Available(fettuccineAlfredo, 4);

    //Step 3: Order a bunch of dishes
    caesarAvailable.OrderItem("John");
    caesarAvailable.OrderItem("Sally");
    caesarAvailable.OrderItem("Manush");

    alfredoAvailable.OrderItem("Sally");
    alfredoAvailable.OrderItem("Francis");
    alfredoAvailable.OrderItem("Venkat");
    alfredoAvailable.OrderItem("Diana");

    //There won't be enough for this order.
    alfredoAvailable.OrderItem("Dennis");

    caesarAvailable.Display();
    alfredoAvailable.Display();

    Console.ReadKey();
}
```

In this example, Dennis won't get the fettuccine alfredo he ordered, because by the time he

gets to order it, we've run out of ingredients to prepare it with.



Will I Ever Use This Pattern?

Maybe? I haven't personally used it outside of demos, but I can see it being useful in many circumstances. I'm also wondering if maybe ASP.NET MVC's attributes count as usage of the Decorator pattern, but I'm not sure.

Summary

The Decorator pattern seeks to dynamically add functionality to instances of an object at runtime, without needing to change the definition of the instance's class. This is especially useful in scenarios where different instances of the same object might behave differently (such as dishes in a restaurant or items in a library).

Day 18: Chain of Responsibility

What Is This Pattern?

The Chain of Responsibility design pattern seeks to avoid coupling a request to a particular receiver by giving more than one object a chance to handle a particular request.

In essence, we pass an object along a "chain" of potential handlers for that object until one of the handlers deals with the request.

The Rundown

- **Type:** Behavioral
- **Useful?** 2/5 (Uncommon)
- **Good For:** Allowing multiple distinct objects to have a chance to process a request.
- **Example Code:** On GitHub
- **Blog Post:** "Chain of Responsibility - The Daily Design Pattern"

The Participants

- The **Handler** defines an interface for handling requests.
- The **ConcreteHandler** objects can each handle a request, and can access their

successor object.
- The **Client** initiates the request to a ConcreteHandler object.

A Delicious Example

We're going to stick with the restaurant example from Day 16: Proxy and Day 17: Decorator. For this day's demo, let's think about how a professional restaurant kitchen might acquire the equipment it needs to operate on a day-to-day basis.

In a given restaurant, like any professional operation, there is probably a hierarchy in which the people who work at that establishment are placed.

When a kitchen needs supplies, the Head Chef of that kitchen is likely to be the first one to notice. So, s/he might need to file a purchase request with his/her bosses in order to obtain new equipment, such as knives or cutting boards or even larger items like ovens.

In our kitchen specifically, the purchase request system operates like this:

1. The Head Chef has implicit approval to purchase any item which is less than $1000 USD.
2. If the total amount of the purchase is greater than that but less than $2500, the Head Chef must get the restaurant's Purchasing Manager's approval for the purchase.
3. If the total amount of the purchase is greater than $2500 but less than $10000, then the head chef must get the approval of the restaurant's General Manager to make the purchase.
4. Finally, if the purchase amount is greater than $10000, the General Manager will call an executive meeting to determine if they need to make the purchase requested.

There's a hierarchy in play here: Head Chef answers to the Purchasing Manager, who answers to the General Manager. We can model this purchasing system using the Chain of Responsibility design pattern.

Firstly, let's model the object that represents the purchase order itself.

```
/// <summary>
/// The details of the purchase request.
/// </summary>
class PurchaseOrder
{
    // Constructor
    public PurchaseOrder(int number, double amount, double price, string name)
```

```
    {
        RequestNumber = number;
        Amount = amount;
        Price = price;
        Name = name;

        Console.WriteLine("Purchase request for " + name
                          + " (" + amount
                          + " for $" + price.ToString()
                          + ") has been submitted.");
    }

    public int RequestNumber { get; set; }
    public double Amount { get; set; }
    public double Price { get; set; }
    public string Name { get; set; }
}
```

With that in place, let's now write an abstract class `Approver`,which is our **Handler** participant. This represents any person in the chain who can approve requests.

```
/// <summary>
/// The Handler abstract class.
/// Every class which inherits from this will be responsible
/// for a kind of request for the restaurant.
/// </summary>
abstract class Approver
{
    protected Approver Supervisor;

    public void SetSupervisor(Approver supervisor)
    {
        this.Supervisor = supervisor;
    }

    public abstract void ProcessRequest(PurchaseOrder purchase);
}
```

Now we can implement our **ConcreteHandler** objects: one for each person in the chain.

```
/// <summary>
/// A concrete Handler class
/// </summary>
class HeadChef : Approver
{
    public override void ProcessRequest(PurchaseOrder purchase)
    {
        if (purchase.Price < 1000)
        {
            Console.WriteLine("{0} approved purchase request #{1}",
                this.GetType().Name, purchase.RequestNumber);
        }
        else if (Supervisor != null)
        {
```

```
            Supervisor.ProcessRequest(purchase);
        }
    }
}

/// <summary>
/// A concrete Handler class
/// </summary>
class PurchasingManager : Approver
{
    public override void ProcessRequest(PurchaseOrder purchase)
    {
        if (purchase.Price < 2500)
        {
            Console.WriteLine("{0} approved purchase request #{1}",
                this.GetType().Name, purchase.RequestNumber);
        }
        else if (Supervisor != null)
        {
            Supervisor.ProcessRequest(purchase);
        }
    }
}

/// <summary>
/// A concrete Handler class
/// </summary>
class GeneralManager : Approver
{
    public override void ProcessRequest(PurchaseOrder purchase)
    {
        if (purchase.Price < 10000)
        {
            Console.WriteLine("{0} approved purchase request #{1}",
                this.GetType().Name, purchase.RequestNumber);
        }
        else
        {
            Console.WriteLine(
                "Purchase request #{0} requires an executive meeting!",
                purchase.RequestNumber);
        }
    }
}
```

Notice that each person in the hierarchy (e.g. each link in the chain) can call its own supervisor to make a determination as to whether or not the item can be purchased. This is part of the Chain of Responsibility design pattern: each link is aware of its own successor.

Finally, we need a **Client** participant, which (as is so often the case in this boo) is our `Main()` method.

```
static void Main(string[] args)
```

```
{
    //Create the chain links
    Approver jennifer = new HeadChef();
    Approver mitchell = new PurchasingManager();
    Approver olivia = new GeneralManager();

    //Create the chain
    jennifer.SetSupervisor(mitchell);
    mitchell.SetSupervisor(olivia);

    //Generate and process purchase requests
    PurchaseOrder p = new PurchaseOrder(1, 20, 69, "Spices");
    jennifer.ProcessRequest(p);

    p = new PurchaseOrder(2, 300, 1389, "Fresh Veggies");
    jennifer.ProcessRequest(p);

    p = new PurchaseOrder(3, 500, 4823.99, "Beef");
    jennifer.ProcessRequest(p);

    p = new PurchaseOrder(4, 4, 12099, "Ovens");
    jennifer.ProcessRequest(p);

    Console.ReadKey();
}
```

Notice that all requests initially flow to Jennifer, the head chef, but if the request's total price is greater than certain amounts then the requets automatically flow to Mitchell the purchasing manager or Olivia the general manager. The output of this example project can be seen in the screenshot on this page.

Each request flows through the chain until a link handles it. That's the definition of the Chain of Responsibility design pattern!

Will I Ever Use This Pattern?

I'm going to say **not often**. I personally haven't used it at all, but I can see why it would be useful in situations where there's a hierarchy of objects and each one could handle a particular request. Particularly in corporate purchasing scenarios, where a request generally has to filter through many levels of managers, I could see this pattern being extremely useful.

Summary

The Chain of Responsibility pattern allows for multiple objects in a chain to make a pass at handling a request object. The request flows through the chain until a link in the chain

handles it, and each link can handle the request in an entirely different fashion from the other links.

Day 19: Visitor

What Is This Pattern?

The Visitor pattern lets us operate on objects by representing that operation as an object unto itself. Thereby, we can operate on said objects without changing the classes or definitions of those objects.

This pattern is particularly useful when, for one reason or another, we cannot modify or refactor existing classes but need to change their behavior.

The Rundown

- **Type:** Behavioral
- **Useful?** 1/5 (Rarely)
- **Good For:** Changing the behavior of an object without changing their class definition.
- **Example Code:** On GitHub
- **Blog Post:** "Visitor - The Daily Design Pattern"

The Participants

- The **Visitor** declares an operation for each of ConcreteElement in the object structure.
- The **ConcreteVisitor** implements each operation defined by the Visitor. Each operation implements a fragment of the algorithm needed for that object.
- The **Element** defines an Accept operation which takes a Visitor as an argument.
- The **ConcreteElement** implements the Accept operation defined by the Element.
- The **ObjectStructure** can enumerate its elements and may provide a high-level interface to allow the Visitor to visit its elements.

A Delicious Example

Let's return to our restaurant example from the past three days' demos to properly model the Visitor design pattern.

Our restaurant has become very successful, with full tables and raving critical reviews, and our GM has decided he wants to do something special to show his employees

that they are appreciated.

Specifically, the GM and the rest of the upper management has decided that it's time to reward our hard-working employees by giving them raises and extra time off. To do this, we need to update the employees' records in our HR system. Problem is, the classes which represent our employees are already created and, for whatever reason, cannot be changed.

Let's first define our immovable **Element** and **ConcreteElement** participants, representing employees of our restaurant. We need to implement a Visitor which will visit these employee records and modify their salary and paid time off accordingly.

```
/// <summary>
/// The Element abstract class.
/// All this does is define an Accept operation,
/// which needs to be implemented by any class that can be visited.
/// </summary>
abstract class Element
{
    public abstract void Accept(IVisitor visitor);
}

/// <summary>
/// The ConcreteElement class,
/// which implements all operations defined by the Element.
/// </summary>
class Employee : Element
{
    public string Name { get; set; }
    public double AnnualSalary { get; set; }
    public int PaidTimeOffDays { get; set; }

    public Employee(string name, double annualSalary, int paidTimeOffDays)
    {
        Name = name;
        AnnualSalary = annualSalary;
        PaidTimeOffDays = paidTimeOffDays;
    }

    public override void Accept(IVisitor visitor)
    {
        visitor.Visit(this);
    }
}
```

We can now being writing the Visitor which will modify our employee records. Let's define an interface `IVisitor` which, no surprise, is also our **Visitor** participant:

```
/// <summary>
/// The Visitor interface,
/// which declares a Visit operation for each ConcreteVisitor to implement.
/// </summary>
interface IVisitor
{
    void Visit(Element element);
```

```
}
```

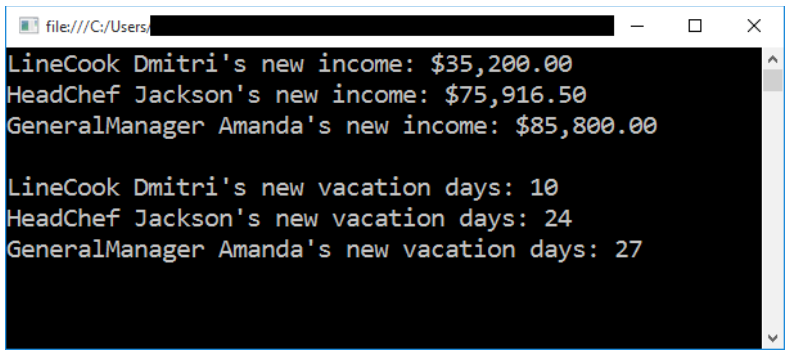
Now we need our **ConcreteVisitor** participants, one for each detail about the employee records that we want to change.

```
/// <summary>
/// A Concrete Visitor class.
/// </summary>
class IncomeVisitor : IVisitor
{
    public void Visit(Element element)
    {
        Employee employee = element as Employee;

        // We've had a great year, so 10% pay raises for everyone!
        employee.AnnualSalary *= 1.10;
        Console.WriteLine("{0} {1}'s new income: {2:C}",
                          employee.GetType().Name,
                          employee.Name,
                          employee.AnnualSalary);
    }
}

/// <summary>
/// A Concrete Visitor class
/// </summary>
class PaidTimeOffVisitor : IVisitor
{
    public void Visit(Element element)
    {
        Employee employee = element as Employee;

        // And because you all helped have such a great year,
        // all my employees get three extra paid time off days each!
        employee.PaidTimeOffDays += 3;
        Console.WriteLine("{0} {1}'s new vacation days: {2}",
                          employee.GetType().Name,
                          employee.Name,
                          employee.PaidTimeOffDays);
    }
}
```

Finally, we need classes to represent all of our employees as a group and individually. The aggregate collection of employees is our **ObjectStructure** participant:

```
/// <summary>
/// The Object Structure class, which is a collection of Concrete Elements.
/// This could be implemented using another pattern such as Composite.
/// </summary>
class Employees
{
    private List<Employee> _employees = new List<Employee>();

    public void Attach(Employee employee)
```

```
    {
        _employees.Add(employee);
    }

    public void Detach(Employee employee)
    {
        _employees.Remove(employee);
    }

    public void Accept(IVisitor visitor)
    {
        foreach (Employee e in _employees)
        {
            e.Accept(visitor);
        }
        Console.WriteLine();
    }
}
class LineCook : Employee
{
    public LineCook() : base("Dmitri", 32000, 7){ }
}

class HeadChef : Employee
{
    public HeadChef() : base("Jackson", 69015, 21){ }
}
class GeneralManager : Employee
{
    public GeneralManager() : base("Amanda", 78000, 24){ }
}
```



When we run the app, we will create a new collection of employees and send visitors to modify their salary and paid time off records. You can see this example in the screenshot to the right.

Will I Ever Use This Pattern?

Probably not, at least not for simple projects. To be honest, I'm tempted to think of this pattern as being the *Burglar* pattern rather than the *Visitor* pattern, since it consists of some heretofore unknown instance of an object showing up, breaking in, rearranging things, and hightailing it out of there.

To be frank, I don't have a lot of first-hand experience with this pattern, but if you think it will help you and your projects, it can't hurt to try it out and see if it works for you.

Summary

The Visitor pattern allows us to modify existing instances of objects without modifying the class they are a part of. All those instances need to do is accept a Visitor object and process its contents. That said, this pattern (IMO) includes a lot of complexity and should be used sparingly.

Day 20: Composite

What Is This Pattern?

The Composite design pattern represents part-whole hierarchies of objects.

("Part-whole hierarchies" is a really fancy way of saying you can represent all or part of a hierarchy by reducing the pieces in said hierarchy down to common components.)

When using this pattern, clients should be able to treat groups of objects in a hierarchy as "the same" even though they can be different. You can do this selectively to parts of the hierarchy, or to the entire hierarchy.

The Rundown

- **Type:** Structural
- **Useful?** 4/5 (Very)
- **Good For:** Treating different objects in a hierarchy as being the same.
- **Example Code:** On GitHub
- **Blog Post:** "Composite - The Daily Design Pattern"

The Participants

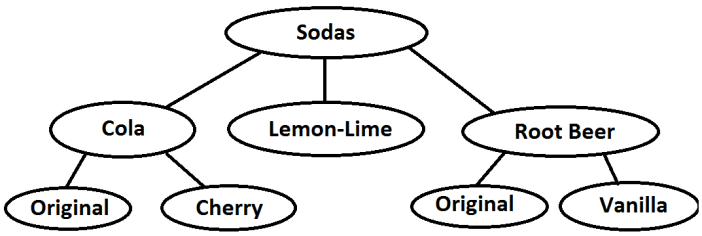
- The **Component** declares an interface for objects in the composition. It also implements behavior that is common to all objects in said composition. Finally, it must implement an interface for adding/removing it's own child components.
- The **Leaves** represent leaf behavior in the composition (a leaf is an object with no children). It also defines primitive behavior for said objects.
- The **Composite** defines behavior for components which have children (contrasting the Leaves). It also stores its child components and implements the add/remove children interface from the Component.
- The **Client** manipulates objects in the composition through the Component interface.

A Delicious Example

We've been writing examples that have all dealt with food, so at this point you're probably a little thirsty. To demonstrate the Composite design pattern, let's talk about a drink dispenser; specifically, the Coca-Cola Freestyle machines like the one in the image to the right that have been popping up at restaurants and movie theatres lately.

For those of you that haven't seen these monstrosities, they're not at all like the regular soft drink dispensers you'll find at restaurants. The regular dispenses have six, or eight, or maybe twelve flavors; the Freestyle machines have potentially *hundreds*. Any flavor of drink that the Coca-Cola company makes in your part of the world, you can order at this machine.

The most interesting part of this device, though, is its interface. The Freestyle wants you to "drill-down" by first selecting a brand (e.g. Coke, Fanta, Sprite, Dasani, etc.) and then selecting a flavor (e.g. Cherry, Vanilla, etc.). In effect, this creates a hierarchy where "Soda" itself is the root Component; the brands are the child Components, and the flavors are Leaves.



A simplified version of this hierarchy might look like the soda flavor tree to the left.

Let's model this hierarchy. For all possible flavors of soda that our machine dispenses, we need to know how many calories each particular flavor has. So, in our abstract class that represents all soft drinks, we need a property for Calories:

```
/// <summary>
/// Soda abstract class
/// </summary>
public abstract class SoftDrink
{
    public int Calories { get; set; }

    public SoftDrink(int calories)
    {
        Calories = calories;
    }
}
```

We also need to implement several **Leaves** for the concrete soda flavors.

```
/// <summary>
/// Leaf class
/// </summary>
public class OriginalCola : SoftDrink
```

```
{
    public OriginalCola(int calories) : base(calories) { }
}

/// <summary>
/// Leaf class
/// </summary>
public class CherryCola : SoftDrink
{
    public CherryCola(int calories) : base(calories) { }
}

/// <summary>
/// Leaf class
/// </summary>
public class OriginalRootBeer : SoftDrink
{
    public OriginalRootBeer(int calories) : base(calories) { }
}

/// <summary>
/// Leaf class
/// </summary>
public class VanillaRootBeer : SoftDrink
{
    public VanillaRootBeer(int calories) : base(calories) { }
}

/// <summary>
/// Leaf class
/// </summary>
public class LemonLime : SoftDrink
{
    public LemonLime(int calories) : base(calories) { }
}
```

Now that we have our Component class and Leaves defined, lets implement the **Composite** participant, which represents objects in the hierarchy which have children. For our decision tree, we have two Composites classes: `Colas` and `RootBeers`.

```
/// <summary>
/// Composite class
/// </summary>
public class Colas
{
    public List<SoftDrink> AvailableFlavors { get; set; }

    public Colas()
    {
        AvailableFlavors = new List<SoftDrink>();
    }
}

/// <summary>
```

```
/// Composite class
/// </summary>
public class RootBeers
{
    public List<SoftDrink> AvailableFlavors { get; set; }

    public RootBeers()
    {
        AvailableFlavors = new List<SoftDrink>();
    }
}
```

Now, let's imagine we need to report the amount of calories in each of our flavors to our customers. The customers doesn't care about our hierarchy, they just wants to know how many calories each flavor has.

Since we've implemented the Composite design pattern, we can provide this data easily in our **Component** participant (which represents the soda dispenser itself):

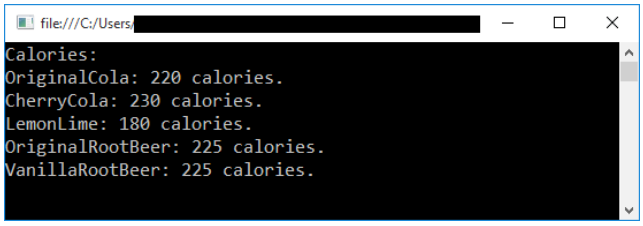
```
/// <summary>
/// The Component class
/// </summary>
public class SodaDispenser
{
    public Colas Colas { get; set; }
    public LemonLime LemonLime { get; set; }
    public RootBeers RootBeers { get; set; }

    public SodaDispenser()
    {
        Colas = new Colas();
        LemonLime = new LemonLime(190);
        RootBeers = new RootBeers();
    }

    /// <summary>
    /// Returns all available flavors and display their calories
    /// </summary>
    public void DisplayCalories()
    {
        var sodas = new Dictionary<string, int>();
        foreach (var cola in Colas.AvailableFlavors)
        {
            sodas.Add(cola.GetType().Name, cola.Calories);
        }
        sodas.Add(LemonLime.GetType().Name, LemonLime.Calories);

        foreach (var rootbeer in RootBeers.AvailableFlavors)
        {
            sodas.Add(rootbeer.GetType().Name, rootbeer.Calories);
        }

        Console.WriteLine("Calories:");
        foreach (var soda in sodas)
```

```
        {
            Console.WriteLine(soda.Key +": "
                              + soda.Value.ToString() + " calories.");
        }
    }
}
```

Finally, our `Main()` method shows how we might initialize a `SodaDispenser` instance with several hierarchical flavors and then display all of the calories for each flavor:

```
static void Main(string[] args)
{
    SodaDispenser fountain = new SodaDispenser();
    fountain.Colas.AvailableFlavors.Add(new OriginalCola(220));
    fountain.Colas.AvailableFlavors.Add(new CherryCola(230));
    fountain.LemonLime.Calories = 180;
    fountain.RootBeers.AvailableFlavors.Add(new OriginalRootBeer(225));
    fountain.RootBeers.AvailableFlavors.Add(new VanillaRootBeer(225));
    fountain.DisplayCalories();

    Console.ReadKey();
}
```

When we run the sample app, we'll see output similar to the screenshot to the right.



Will I Ever Use This Pattern?

Will you ever have hierarchical data? If so, **probably yes**. The key part of this pattern is that you can treat different objects as the same, provided you set up the appropriate interfaces and abstracts.

Summary

The Composite pattern takes objects in a hierarchy and allows clients to treat different parts of that hierarchy as being the same. Then, among other things, you can "flatten" all or part of the hierarchy to get only the data that's common to all of the parts.

Day 21: Mediator

What Is This Pattern?

The Mediator design pattern defines an object which encapsulates how a set of objects interact with each other.

You can think of a Mediator object as a kind of traffic-coordinator; it directs traffic to

appropriate parties based on its own state or outside values. Further, Mediator promotes loose coupling (a good thing!) by keeping objects from referring to each other explicitly.

The Rundown

- **Type:** Behavioral
- **Useful?** 2/5 (Uncommon)
- **Good For:** Defining how objects interact and communicate with each other.
- **Example Code:** On GitHub
- **Blog Post:** "Mediator - The Daily Design Pattern"

The Participants

- The **Mediator** defines an interface for communicating with Collegue objects.
- The **Colleague** classes each know what Mediator is responsible for them and communicates with said Mediator whenever it would have otherwise communicated directly with another Colleague.
- The **ConcreteMediator** classes implement behavior to coordinate Colleague objects. Each ConcreteMediator knows what its constituent Colleague classes are.

A Delicious Example

To demo the Mediator pattern, let's consider the snack bars in your local movie theatre.

Movie theatres, relative to other kinds of buildings, tend to take up a lot of ground space. A particular cinema that's not too far from me has 25 screens spread out over three different "sections" of the theatre. Each of these sections has their own snack bar, from which we gluttonous patrons can order salty snacks and sugary drinks to our increasingly-stressed heart's content.

But selling concessions to hungry movie-goers requires supplies, and sometimes the different snack bars might run out of said supplies. Let's imagine a system in which the different concession stands can talk to each other, communicating what supplies they need and who might have them (in short, a chat system for movie snack bars). We can model this system using the Mediator pattern.

First, we'll need our **Mediator** interface, which defines a method by which the snack bars can talk to each other:

```
/// <summary>
/// The Mediator interface, which defines a send message
```

```
/// method which the concrete mediators must implement.
/// </summary>
interface Mediator
{
    void SendMessage(string message, ConcessionStand concessionStand);
}
```

We also need an abstract class to represent the **Colleagues** that will be talking to one another:

```
/// <summary>
/// The Colleague abstract class, representing an entity
/// involved in the conversation which should receive messages.
/// </summary>
abstract class ConcessionStand
{
    protected Mediator mediator;

    public ConcessionStand(Mediator mediator)
    {
        this.mediator = mediator;
    }
}
```

Now let's implement the different Colleagues. In this case, we'll pretend our movie theatre has two snack bars: one in the northern part of the theatre and one in the southern part.

```
/// <summary>
/// A Concrete Colleague class
/// </summary>
class NorthConcessionStand : ConcessionStand
{
    // Constructor
    public NorthConcessionStand(Mediator mediator) : base(mediator)
    {
    }

    public void Send(string message)
    {
        Console.WriteLine("North Concession Stand sends message: " + message);
        mediator.SendMessage(message, this);
    }

    public void Notify(string message)
    {
        Console.WriteLine("North Concession Stand gets message: "  + message);
    }
}

/// <summary>
/// A Concrete Colleague class
/// </summary>
class SouthConcessionStand : ConcessionStand
{
```

```
    public SouthConcessionStand(Mediator mediator) : base(mediator)
    {
    }

    public void Send(string message)
    {
        Console.WriteLine("South Concession Stand sends message: " + message);
        mediator.SendMessage(message, this);
    }

    public void Notify(string message)
    {
        Console.WriteLine("South Concession Stand gets message: " + message);
    }
}
```

Note that each Colleague must be aware of the Mediator that is mediating the colleague's messages.

Finally, we can implement the **ConcreteMediator** class, which will keep a reference to each Colleague and manage communication between them.

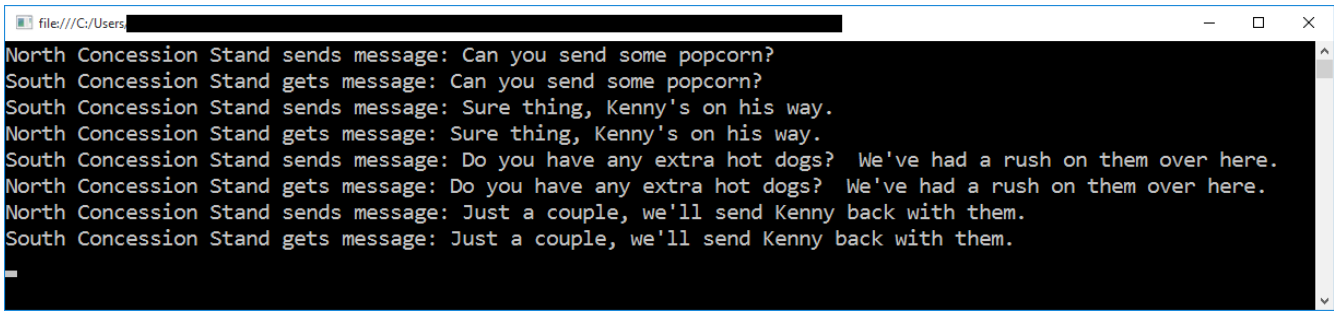
```
/// <summary>
/// The Concrete Mediator class, which implement the send message method
/// and keeps track of all participants in the conversation.
/// </summary>
class ConcessionsMediator : Mediator
{
    private NorthConcessionStand _northConcessions;
    private SouthConcessionStand _southConcessions;

    public NorthConcessionStand NorthConcessions
    {
        set { _northConcessions = value; }
    }

    public SouthConcessionStand SouthConcessions
    {
        set { _southConcessions = value; }
    }

    public void SendMessage(string message, ConcessionStand colleague)
    {
        if (colleague == _northConcessions)
        {
            _southConcessions.Notify(message);
        }
        else
        {
            _northConcessions.Notify(message);
        }
    }
}
```

In our `Main()` method, we can use our newly-written Mediator to simulate a chat conversation between the two snack bars.

Suppose that one of the snack bars has run out of popcorn, and needs to know if the other has extra that they're not using:

```
static void Main(string[] args)
{
    ConcessionsMediator mediator = new ConcessionsMediator();

    NorthConcessionStand leftKitchen = new NorthConcessionStand(mediator);
    SouthConcessionStand rightKitchen = new SouthConcessionStand(mediator);

    mediator.NorthConcessions = leftKitchen;
    mediator.SouthConcessions = rightKitchen;

    leftKitchen.Send("Can you send some popcorn?");
    rightKitchen.Send("Sure thing, Kenny's on his way.");

    rightKitchen.Send("Do you have any extra hot dogs?  We've had a rush on them
over here.");
    leftKitchen.Send("Just a couple, we'll send Kenny back with them.");

    Console.ReadKey();
}
```

If we run this app, we'll see a conversation between the two concession stands (send/receive messages added to clearly show who sent what):



Will I Ever Use This Pattern?

To be honest, **probably not**. It's only useful in specific scenarios (e.g. chat systems and the like) and may not be terribly applicable to other types of projects. But, for those kinds of systems, I could see Mediator being extremely useful.

Summary

The Mediator pattern encapsulates an object which represents how other objects communicate with one another. By doing so, it enables the Mediator to "stand between" communicating objects and control their communications.

Day 22: Command

What Is This Pattern?

The Command design pattern encapsulates a request as an object, thereby allowing us developers to treat that request differently based upon what class receives said command. Further, it enables much more complex architectures, and even enables operations such as undo/redo.

The Chain of Responsibility pattern fits well with the Command pattern, as the former can use objects of the latter to represent its requests.

The Rundown

- **Type:** Behavioral
- **Useful?** 4/5 (Very)
- **Good For:** Encapsulating requests as objects so that they can be processed differently by different receivers.
- **Example Code:** On GitHub
- **Blog Post:** "Command - The Daily Design Pattern"

The Participants

- The **Command** declares an interface for executing an operation.
- The **ConcreteCommand** defines a binding between a Receiver and an action.
- The **Client** creates a ConcreteCommand object and sets its receiver.
- The **Invoker** asks the command to carry out its request.
- The **Receiver** knows how to perform the operations associated with carrying out the request.

A Delicious Example

Since just defining the Participants doesn't do a very thorough job of explaining what this pattern is all about, let's build a demo project to show how the Command design pattern truly works.

In this project, we'll model a system in which we can create an order for a fast food restaurant, and add, remove, and modify items in the order using the Command design pattern.

To begin building our demo, let's first create a class which represents an item being ordered.

```
/// <summary>
/// Represents an item being ordered from this restaurant.
/// </summary>
public class MenuItem
```

```
{
    public string Name { get; set; }
    public int Amount { get; set; }
    public double Price { get; set; }

    public MenuItem(string name, int amount, double price)
    {
        Name = name;
        Amount = amount;
        Price = price;
    }

    public void Display()
    {
        Console.WriteLine("\nName: " + Name);
        Console.WriteLine("Amount: " + Amount.ToString());
        Console.WriteLine("Price: $" + Price.ToString());
    }
}
```

Since those items will be ordered by a patron of the restaurant, let's create a `Patron` object which will also be our **Invoker** participant. It just so happens that our implementation of the Invoker also includes a Factory Method (from all the way back in Day 1: Factory Method):

```
/// <summary>
/// The Invoker class
/// </summary>
public class Patron
{
    private OrderCommand _orderCommand;
    private MenuItem _menuItem;
    private FastFoodOrder _order;

    public Patron()
    {
        _order = new FastFoodOrder();
    }

    public void SetCommand(int commandOption)
    {
        _orderCommand = new CommandFactory().GetCommand(commandOption);
    }

    public void SetMenuItem(MenuItem item)
    {
        _menuItem = item;
    }

    public void ExecuteCommand()
    {
        _order.ExecuteCommand(_orderCommand, _menuItem);
    }

    public void ShowCurrentOrder()
```

```
    {
        _order.ShowCurrentItems();
    }
}

public class CommandFactory
{
    //Factory method
    public OrderCommand GetCommand(int commandOption)
    {
        switch (commandOption)
        {
            case 1:
                return new AddCommand();
            case 2:
                return new ModifyCommand();
            case 3:
                return new RemoveCommand();
            default:
                return new AddCommand();
        }
    }
}
```

Note that the `Patron` keeps a reference to an instance of `FastFoodOrder`, which is our Receiver participant and is implemented like so:

```
/// <summary>
/// The Receiver
/// </summary>
public class FastFoodOrder
{
    public List<MenuItem> currentItems { get; set; }
    public FastFoodOrder()
    {
        currentItems = new List<MenuItem>();
    }

    public void ExecuteCommand(OrderCommand command, MenuItem item)
    {
        command.Execute(this.currentItems, item);
    }

    public void ShowCurrentItems()
    {
        foreach(var item in currentItems)
        {
            item.Display();
        }
        Console.WriteLine("------------------------");
    }
}
```

The `FastFoodOrder` keeps track of all items in the order, so that when commands arrive at it,

it can process those commands using its own list of items.

Speaking of the commands, we can now write up the base **Command** participant:

```
/// <summary>
/// The Command abstract class
/// </summary>
public abstract class OrderCommand
{
    public abstract void Execute(List<MenuItem> order, MenuItem newItem);
}
```

Now we can also implement several **ConcreteCommand** objects:

```
/// <summary>
/// A concrete command
/// </summary>
public class AddCommand : OrderCommand
{
    public override void Execute(List<MenuItem> currentItems, MenuItem newItem)
    {
        currentItems.Add(newItem);
    }
}

/// <summary>
/// A concrete command
/// </summary>
public class RemoveCommand : OrderCommand
{
    public override void Execute(List<MenuItem> currentItems, MenuItem newItem)
    {
        currentItems.Remove(currentItems.Where(x=>x.Name ==
newItem.Name).First());
    }
}

/// <summary>
/// A concrete command
/// </summary>
public class ModifyCommand : OrderCommand
{
    public override void Execute(List<MenuItem> currentItems, MenuItem newItem)
    {
        var item = currentItems.Where(x => x.Name == newItem.Name).First();
        item.Price = newItem.Price;
        item.Amount = newItem.Amount;
    }
}
```

Now that we've got all the pieces in place, let's create our **Client** participant, which creates a ConcreteCommand and sets the receiver; in this case, we will add several items to our order, then delete an item and change another item.

```
static void Main(string[] args)
{
    Patron patron = new Patron();
    patron.SetCommand(1 /*Add*/);
    patron.SetMenuItem(new MenuItem("French Fries", 2, 1.99));
    patron.ExecuteCommand();

    patron.SetCommand(1 /*Add*/);
    patron.SetMenuItem(new MenuItem("Hamburger", 2, 2.59));
    patron.ExecuteCommand();

    patron.SetCommand(1 /*Add*/);
    patron.SetMenuItem(new MenuItem("Drink", 2, 1.19));
    patron.ExecuteCommand();

    patron.ShowCurrentOrder();

    //Remove the french fries
    patron.SetCommand(3 /*Add*/);
    patron.SetMenuItem(new MenuItem("French Fries", 2, 1.99));
    patron.ExecuteCommand();

    patron.ShowCurrentOrder();

    //Now we want 4 hamburgers rather than 2
    patron.SetCommand(2 /*Add*/);
    patron.SetMenuItem(new MenuItem("Hamburger", 4, 2.59));
    patron.ExecuteCommand();

    patron.ShowCurrentOrder();

    Console.ReadKey();
}
```

As the orders are processed by the Receiver (the `FastFoodOrder` instance), the contents of the order changes. The output for this sample project can be seen in the screenshot to the right.

Will I Ever Use This Pattern?

I have, but you will **probably not**, unless you are using more complex architectures. In my case, we're building an app using command query responsibility segregation and event sourcing, two complex architectures which, together, are implementations of the Command design pattern blown up to support large, intricate projects. The Command design pattern is an extremely useful pattern, but invokes a lot of complexity (more so than many of the other design patterns) so use this design pattern with the requisite caution.

Summary

The Command design pattern seeks to encapsulate commands as objects and allow different receivers to process them, according to the receivers' own design.

Day 23: Wrapup

You made it! Over the past 22 days, you've learned and demoed 22 software design patterns, and hopefully understand a little more about what kinds of problems these patterns solve and how to use them properly.

Don't forget to check out this book's GitHub repository; I'd be happy to consider any suggestions made to improve my sample code.

Finally, if you have any comments you'd like to leave about this series and how it helped you (or how I can improve it), check out the series index page on my blog.

Thanks for reading, and as always, Happy Coding!

Appendix A: Patterns Are Tools, Not Goals

The following is a blog post I wrote before I started the Daily Design Pattern series which details how I believe design patterns should be used. You can read the post in its original format over on my blog.

I went through a phase earlier in my career where I thought design patterns were the be-all, end-all of software design. Any system which I needed to design started with the applicable patterns: Factories, Repositories, Singletons, you name it. Invariably, though, these systems were difficult to maintain and more difficult to explain to my coworkers, and I never quite seemed to put two and two together. The fact was that I just didn't understand them the way I thought I did.

Five years later, I've now been researching these same design patterns for a presentation I'm giving at my day job, the goal of which is to demonstrate how said patterns help us maintain our projects in a C#/.NET environment. I've built and documented several examples for Adapter, Facade, Abstract Factory, etc. They are making sense to me, I'm definitely understand them more thoroughly, but there's something still seems a little... off.

To be clear, I've never read the Gang of Four book these patterns are defined in, so it's possible there's reasoning in the book that would alleviate my concerns. In fact, all of my knowledge about these patterns has come from online resources such as Do Factory. And yet the more I understand them, the more I believe that design patterns are not goals which we should strive for.

Take the Adapter pattern as an example. Adapter strives to provide an abstraction over an interface such that a client which expects a *different* interface can still access the old one. Imagine that we have a legacy API which looks something like this (taken directly from Do Factory):

```
class Compound
{
    protected string _chemical;
    protected float _boilingPoint;
    protected float _meltingPoint;
    protected double _molecularWeight;
    protected string _molecularFormula;

    // Constructor
    public Compound(string chemical)
    {
        this._chemical = chemical;
    }

    public virtual void Display()
    {
        Console.WriteLine("\nCompound: {0} ------ ", _chemical);
    }
}
```

```
class ChemicalDatabank
{
    // The databank 'legacy API'
    public float GetCriticalPoint(string compound, string point)
    {
        // Melting Point
        if (point == "M")
        {
            switch (compound.ToLower())
            {
                case "water": return 0.0f;
                case "benzene": return 5.5f;
                case "ethanol": return -114.1f;
                default: return 0f;
            }
        }
        // Boiling Point
        else
        {
            switch (compound.ToLower())
            {
                case "water": return 100.0f;
                case "benzene": return 80.1f;
                case "ethanol": return 78.3f;
                default: return 0f;
            }
        }
    }

    public string GetMolecularStructure(string compound)
    {
        switch (compound.ToLower())
        {
            case "water": return "H20";
            case "benzene": return "C6H6";
            case "ethanol": return "C2H5OH";
            default: return "";
        }
    }

    public double GetMolecularWeight(string compound)
    {
        switch (compound.ToLower())
        {
            case "water": return 18.015;
            case "benzene": return 78.1134;
            case "ethanol": return 46.0688;
            default: return 0d;
        }
    }
}
```

However, our new system expects Critical Point, Molecular Weight and Molecular Structure to

be properties of an object called `RichCompound`, rather than queries to an API, so the Adapter patterns says we should do this:

```
class RichCompound : Compound
{
    private ChemicalDatabank _bank;

    // Constructor
    public RichCompound(string name)
        : base(name)
    {
    }

    public override void Display()
    {
        // The Adaptee
        _bank = new ChemicalDatabank();

        _boilingPoint = _bank.GetCriticalPoint(_chemical, "B");
        _meltingPoint = _bank.GetCriticalPoint(_chemical, "M");
        _molecularWeight = _bank.GetMolecularWeight(_chemical);
        _molecularFormula = _bank.GetMolecularStructure(_chemical);

        base.Display();
        Console.WriteLine(" Formula: {0}", _molecularFormula);
        Console.WriteLine(" Weight : {0}", _molecularWeight);
        Console.WriteLine(" Melting Pt: {0}", _meltingPoint);
        Console.WriteLine(" Boiling Pt: {0}", _boilingPoint);
    }
}
```

The `RichCompound` class is therefore an adapter for the legacy API: it converts what was service calls into properties. That way our new system can use the class it expects and the legacy API doesn't need to go through a rewrite.

Here's the problem I have with design patterns like these: they seem to be something that should occur organically rather than intentionally. We shouldn't directly target having any of these patterns in our code, but we should know what they are so that if we accidentally create one, we can better describe it to others.

In other words, **if you ever find yourself thinking, "I know, I'll use a design pattern" before writing any code, you're doing it wrong**.

What patterns don't help with is the initial design of a system. In this phase, the only thing you should be worried about is how to faithfully and correctly implement the business rules and procedures. Following that, you can create a "correct" architecture, for whatever correct means to you, your business, your clients, and your code standards. Patterns don't help during this phase because they artificially restrict what you think your code can do. If you start seeing Adapters everywhere, it becomes much harder to think of a structure that may not have a name but would fit better in your app's architecture.

I see design patterns as tools for refactoring and communication. By learning what they are and what they are used for, we can more quickly refactor troublesome projects and more thoroughly understand unfamiliar ones. If we see something that we can recognize as, say, the Composite pattern, we can then look for the individual pieces of the pattern (the tree structure, the leaves and branches, the common class between them) and more rapidly learn why this structure was used. **Recognizing patterns help you uncover *why* the code was structured in a certain way**.

Of course, this strategy can backfire if the patterns were applied inappropriately in the first place. If someone used a Singleton where it didn't make sense, we might be stuck, having assumed that they knew what they were doing. Hence, don't use patterns at when beginning your design of a project, use them after you've got a comprehensive structure in place, and only where they make sense. Shoehorning patterns into places where they don't make sense is a recipe for unmaintainable projects.

Software design patterns are tools, not goals. Use them as such, and only when you actually need them, not before.

Appendix B: Image Credits

- The Dagwood image (Factory Method) is from Wikimedia, used under license.
- The waitress image (Façade) is from Wikimedia, used under license.
- The jelly beans image (Iterator) is from Wikimedia, used under license.
- The vegetable market image (Observer) is from Wikimedia, used under license.
- The sliders image (Flyweight) is from Wikimedia, used under license.
- The grilling steaks image (State) is from Wikimedia, used under license.
- The grilled food image (Strategy) is from Wikimedia, used under license.
- The kitchen layout image (Chain of Responsibility) is from Wikimedia, used under license.
- The full restaurant image (Visitor) is from Flickr, used under license.
- The concession stand image (Mediator) is from Wikimedia, used under license.
- The fast food counter image (Command) is from Wikimedia, used under license.