<h3 align="center">

Adaptation of <a href="https://github.com/kamranahmedse/design-patterns-for-humans">Design Patterns for Humans</a>  to C#

</h3>

<p align="center"><sub>All the explanation for design patterns stays the same, with minor changes.</sub></p>

****

<p align="center">

🎉 Ultra-simplified explanation to design patterns! 🎉

</p>

<p align="center">

A topic that can easily make anyone's mind wobble. Here I try to make them stick in to your mind (and maybe mine) by explaining them in the <i>simplest</i> way possible.

</p>

<p align="center">

You can find full length examples for code snippets used in this article <a href="https://github.com/anupavanm/csharp-design-patterns-for-humans-examples">here.</a>

</p>

****

🚀 Introduction

=================

Design patterns are solutions to recurring problems; **guidelines on how to tackle certain problems**. They are not classes, packages or libraries that you can plug into your application and wait for the magic to happen. These are, rather, guidelines on how to tackle certain problems in certain situations.

> Design patterns are solutions to recurring problems; guidelines on how to tackle certain problems

Wikipedia describes them as

> In software engineering, a software design pattern is a general reusable solution to a commonly occurring problem within a given context in software design. It is not a finished design that can be transformed directly into source or machine code. It is a description or template for how to solve a problem that can be used in many different situations.

⚠️ Be Careful

-----------------

- Design patterns are not a silver bullet to all your problems.

- Do not try to force them; bad things are supposed to happen, if done so. Keep in mind that design patterns are solutions **to** problems, not solutions **finding** problems; so don't overthink.

- If used in a correct place in a correct manner, they can prove to be a savior; or else they can result in a horrible mess of a code.

> Also note that the code samples below are in C#-7, however this shouldn't stop you because the concepts are same anyways. Plus the **support for other languages is underway**.

Types of Design Patterns

-----------------

* [Creational](#creational-design-patterns)

* [Structural](#structural-design-patterns)

* [Behavioral](#behavioral-design-patterns)

Creational Design Patterns

==========================

In plain words

> Creational patterns are focused towards how to instantiate an object or group of related objects.

Wikipedia says

> In software engineering, creational design patterns are design patterns that deal with object creation mechanisms, trying to create objects in a manner suitable to the situation. The basic form of object creation could result in design problems or added complexity to the design. Creational design patterns solve this problem by somehow controlling this object creation.

 * [Simple Factory](#-simple-factory)

 * [Factory Method](#-factory-method)

 * [Abstract Factory](#-abstract-factory)

 * [Builder](#-builder)

 * [Prototype](#-prototype)

 * [Singleton](#-singleton)

🏠 Simple Factory

--------------

Real world example

> Consider, you are building a house and you need doors. It would be a mess if every time you need a door, you put on your carpenter clothes and start making a door in your house. Instead you get it made from a factory.

In plain words

> Simple factory simply generates an instance for client without exposing any instantiation logic to the client

Wikipedia says

> In object-oriented programming (OOP), a factory is an object for creating other objects – formally a factory is a function or method that returns objects of a varying prototype or class from some method call, which is assumed to be "new".

**Programmatic Example**

First of all we have a door interface and the implementation

```C#
public interface IDoor
{
    int GetHeight();
    int GetWidth();
}

public class WoodenDoor : IDoor
{
    private int Height { get; set; }
    private int Width { get; set; }

    public WoodenDoor(int height, int width)
    {
        this.Height = height;
        this.Width = width;
    }

    public int GetHeight()
    {
        return this.Height;
    }
    public int GetWidth()
    {
        return this.Width;
    }
}
```

Then we have our door factory that makes the door and returns it

```C#
public static class DoorFactory
{
    public static IDoor MakeDoor(int height, int width)
    {
        return new WoodenDoor(height, width);
    }
}
```

And then it can be used as

```C#
var door = DoorFactory.MakeDoor(80, 30);
Console.WriteLine($"Height of Door : {door.GetHeight()}");
Console.WriteLine($"Width of Door : {door.GetWidth()}");
```

**When to Use?**

When creating an object is not just a few assignments and involves some logic, it makes sense to put it in a dedicated factory instead of repeating the same code everywhere.

🏭 Factory Method

--------------

Real world example

> Consider the case of a hiring manager. It is impossible for one person to interview for each of the positions. Based on the job opening, she has to decide and delegate the interview steps to different people.

In plain words

> It provides a way to delegate the instantiation logic to child classes.

Wikipedia says

> In class-based programming, the factory method pattern is a creational pattern that uses factory methods to deal with the problem of creating objects without having to specify the exact class of the object that will be created. This is done by creating objects by calling a factory method—either specified in an interface and implemented by child classes, or implemented in a base class and optionally overridden by derived classes—rather than by calling a constructor.

**Programmatic Example**

Taking our hiring manager example above. First of all we have an interviewer interface and some implementations for it

```C#
interface IInterviewer
{
    void AskQuestions();
}

class Developer : IInterviewer
{
    public void AskQuestions()
    {
        Console.WriteLine("Asking about design patterns!");
    }
}

class CommunityExecutive : IInterviewer
```

```
{
    public void AskQuestions()
    {
        Console.WriteLine("Asking about community building!");
    }
}
```

Now let us create our `HiringManager`

```C#
abstract class HiringManager
{
    // Factory method
    abstract protected IInterviewer MakeInterviewer();
    public void TakeInterview()
    {
        var interviewer = this.MakeInterviewer();
        interviewer.AskQuestions();
    }
}

```
Now any child can extend it and provide the required interviewer
```C#
class DevelopmentManager : HiringManager
{
    protected override IInterviewer MakeInterviewer()
    {
        return new Developer();
    }
}

class MarketingManager : HiringManager
{
    protected override IInterviewer MakeInterviewer()
    {
        return new CommunityExecutive();
    }
}

```
and then it can be used as

```C#
var devManager = new DevelopmentManager();
devManager.TakeInterview(); //Output : Asking about design patterns!

var marketingManager = new MarketingManager();
marketingManager.TakeInterview();//Output : Asking about community building!

```

**When to use?**

Useful when there is some generic processing in a class but the required sub-class is dynamically decided at runtime. Or putting it in other words, when the client doesn't know what exact sub-class it might need.

🔨 Abstract Factory

----------------

Real world example

> Extending our door example from Simple Factory. Based on your needs you might get a wooden door from a wooden door shop, iron door from an iron shop or a PVC door from the relevant shop. Plus you might need a guy with different kind of specialities to fit the door, for example a carpenter for wooden door, welder for iron door etc. As you can see there is a dependency between the doors now, wooden door needs carpenter, iron door needs a welder etc.

In plain words

> A factory of factories; a factory that groups the individual but related/dependent factories together without specifying their concrete classes.

Wikipedia says

> The abstract factory pattern provides a way to encapsulate a group of individual factories that have a common theme without specifying their concrete classes

**Programmatic Example**

Translating the door example above. First of all we have our `Door` interface and some implementation for it

```C#
interface IDoor {

  void GetDescription();

}
class WoodenDoor : IDoor
```

```
{
  public void GetDescription()
  {
    Console.WriteLine("I am a wooden door");
  }
}

class IronDoor : IDoor
{
  public void GetDescription()
  {
    Console.WriteLine("I am a iron door");
  }
}
```
Then we have some fitting experts for each door type

```C#
interface IDoorFittingExpert
{
  void GetDescription();
}

class Welder : IDoorFittingExpert
{
  public void GetDescription()
  {
    Console.WriteLine("I can only fit iron doors");
  }
}


class Carpenter : IDoorFittingExpert

{

  public void GetDescription()

  {

    Console.WriteLine("I can only fit wooden doors");

  }

}
```

Now we have our abstract factory that would let us make family of related objects i.e. wooden door factory would create a wooden door and wooden door fitting expert and iron door factory would create an iron door and iron door fitting expert

```C#

interface IDoorFactory {

  IDoor MakeDoor();

  IDoorFittingExpert MakeFittingExpert();

}


// Wooden factory to return carpenter and wooden door

class WoodenDoorFactory : IDoorFactory

{

  public IDoor MakeDoor()

  {

    return new WoodenDoor();

  }


  public IDoorFittingExpert MakeFittingExpert()
  {
    return new Carpenter();
  }
}

// Iron door factory to get iron door and the relevant fitting expert
class IronDoorFactory : IDoorFactory
{
  public IDoor MakeDoor()
  {
    return new IronDoor();
  }

  public IDoorFittingExpert MakeFittingExpert()
  {
    return new Welder();
  }
}
```
And then it can be used as
```C#
var woodenDoorFactory = new WoodenDoorFactory();

var woodenDoor = woodenDoorFactory.MakeDoor();
var woodenDoorFittingExpert = woodenDoorFactory.MakeFittingExpert();

woodenDoor.GetDescription(); //Output : I am a wooden door
woodenDoorFittingExpert.GetDescription();//Output : I can only fit woooden doors
```

```
var ironDoorFactory = new IronDoorFactory();


var ironDoor = ironDoorFactory.MakeDoor();
var ironDoorFittingExpert = ironDoorFactory.MakeFittingExpert();


ironDoor.GetDescription();//Output : I am a iron door
ironDoorFittingExpert.GetDescription();//Output : I can only fit iron doors
```

As you can see the wooden door factory has encapsulated the `carpenter` and the `wooden door` also iron door factory has encapsulated the `iron door` and `welder`. And thus it had helped us make sure that for each of the created door, we do not get a wrong fitting expert.

**When to use?**

When there are interrelated dependencies with not-that-simple creation logic involved

👷 Builder
--------------------------------------------

Real world example

> Imagine you are at Hardee's and you order a specific deal, lets say, "Big Hardee" and they hand it over to you without *any questions*; this is the example of simple factory. But there are cases when the creation logic might involve more steps. For example you want a customized Subway deal, you have several options in how your burger is made e.g what bread do you want? what types of sauces would you like? What cheese would you want? etc. In such cases builder pattern comes to the rescue.

In plain words

> Allows you to create different flavors of an object while avoiding constructor pollution. Useful when there could be several flavors of an object. Or when there are a lot of steps involved in creation of an object.

Wikipedia says

> The builder pattern is an object creation software design pattern with the intentions of finding a solution to the telescoping constructor anti-pattern.

Having said that let me add a bit about what telescoping constructor anti-pattern is. At one point or the other we have all seen a constructor like below:

```C#
public Burger(int size, bool cheese, bool pepperoni, bool lettuce, bool tomato)
{
}
```

As you can see; the number of constructor parameters can quickly get out of hand and it might become difficult to understand the arrangement of parameters. Plus this parameter list could keep on growing if you would want to add more options in future. This is called telescoping constructor anti-pattern.

**Programmatic Example**

The sane alternative is to use the builder pattern. First of all we have our burger that we want to make

```C#
class Burger
{
  private int mSize;
  private bool mCheese;
  private bool mPepperoni;
  private bool mLettuce;
  private bool mTomato;
  public Burger(BurgerBuilder builder)
  {
    this.mSize = builder.Size;
    this.mCheese = builder.Cheese;
    this.mPepperoni = builder.Pepperoni;
    this.mLettuce = builder.Lettuce;
    this.mTomato = builder.Tomato;
  }

  public string GetDescription()
  {
    var sb = new StringBuilder();
    sb.Append($"This is {this.mSize} inch Burger. ");
    return sb.ToString();
  }
}
```

And then we have the builder

```C#
class BurgerBuilder {
  public int Size;
  public bool Cheese;
  public bool Pepperoni;
  public bool Lettuce;
  public bool Tomato;

  public BurgerBuilder(int size)
```

```
{
  this.Size = size;
}

public BurgerBuilder AddCheese()
{
  this.Cheese = true;
  return this;
}

public BurgerBuilder AddPepperoni()
{
  this.Pepperoni = true;
  return this;
}

public BurgerBuilder AddLettuce()
{
  this.Lettuce = true;
  return this;
}

public BurgerBuilder AddTomato()
{
  this.Tomato = true;
  return this;
}

public Burger Build()
```

```
 {
   return new Burger(this);
 }
}
```
And then it can be used as:

```C#
var burger = new BurgerBuilder(4).AddCheese()
                              .AddPepperoni()
                              .AddLettuce()
                              .AddTomato()
                              .Build();
Console.WriteLine(burger.GetDescription());
```

**When to use?**

When there could be several flavors of an object and to avoid the constructor telescoping. The key difference from the factory pattern is that; factory pattern is to be used when the creation is a one step process while builder pattern is to be used when the creation is a multi step process.

🐑 Prototype

------------

Real world example

> Remember dolly? The sheep that was cloned! Lets not get into the details but the key point here is that it is all about cloning

In plain words

> Create object based on an existing object through cloning.

Wikipedia says

> The prototype pattern is a creational design pattern in software development. It is used when the type of objects to create is determined by a prototypical instance, which is cloned to produce new objects.

In short, it allows you to create a copy of an existing object and modify it to your needs, instead of going through the trouble of creating an object from scratch and setting it up.

**Programmatic Example**

In C#, it can be easily done using `MemberwiseClone()`

```C#
class Sheep
{
  public string Name { get; set; }

  public string Category { get; set; }

  public Sheep(string name, string category)
  {
    Name = name;
    Category = category;
  }

  public Sheep Clone()
  {
    return MemberwiseClone() as Sheep;
  }
```

}

Then it can be cloned like below

```C#

var original = new Sheep("Jolly", "Mountain Sheep");

Console.WriteLine(original.Name); // Jolly

Console.WriteLine(original.Category); // Mountain Sheep


var cloned = original.Clone();

cloned.Name = "Dolly";

Console.WriteLine(cloned.Name); // Dolly

Console.WriteLine(cloned.Category); // Mountain Sheep

Console.WriteLine(original.Name); // Jolly

```

**When to use?**

When an object is required that is similar to existing object or when the creation would be expensive as compared to cloning.

💍 Singleton

------------

Real world example

> There can only be one president of a country at a time. The same president has to be brought to action, whenever duty calls. President here is singleton.

In plain words

> Ensures that only one object of a particular class is ever created.

Wikipedia says

> In software engineering, the singleton pattern is a software design pattern that restricts the instantiation of a class to one object. This is useful when exactly one object is needed to coordinate actions across the system.

Singleton pattern is actually considered an anti-pattern and overuse of it should be avoided. It is not necessarily bad and could have some valid use-cases but should be used with caution because it introduces a global state in your application and change to it in one place could affect in the other areas and it could become pretty difficult to debug. The other bad thing about them is it makes your code tightly coupled plus mocking the singleton could be difficult.

**Programmatic Example**

To create a singleton, make the constructor private, disable cloning, disable extension and create a static variable to house the instance

```C#
public class President
{
  static President instance;
  // Private constructor
  private President()
  {
    //Hiding the Constructor
  }

  // Public constructor
  public static President GetInstance()
  {
    if (instance == null) {
      instance = new President();
    }
```

    return instance;
  }
}

Then in order to use

```C#
President a = President.GetInstance();
President b = President.GetInstance();

Console.WriteLine(a == b); //Output : true
```

Structural Design Patterns
==========================

In plain words

> Structural patterns are mostly concerned with object composition or in other words how the entities can use each other. Or yet another explanation would be, they help in answering "How to build a software component?"

Wikipedia says

> In software engineering, structural design patterns are design patterns that ease the design by identifying a simple way to realize relationships between entities.

 * [Adapter](#-adapter)
 * [Bridge](#-bridge)
 * [Composite](#-composite)
 * [Decorator](#-decorator)
 * [Facade](#-facade)
 * [Flyweight](#-flyweight)
 * [Proxy](#-proxy)

🔌 Adapter

-------

Real world example

> Consider that you have some pictures in your memory card and you need to transfer them to your computer. In order to transfer them you need some kind of adapter that is compatible with your computer ports so that you can attach memory card to your computer. In this case card reader is an adapter.

> Another example would be the famous power adapter; a three legged plug can't be connected to a two pronged outlet, it needs to use a power adapter that makes it compatible with the two pronged outlet.

> Yet another example would be a translator translating words spoken by one person to another

In plain words

> Adapter pattern lets you wrap an otherwise incompatible object in an adapter to make it compatible with another class.

Wikipedia says

> In software engineering, the adapter pattern is a software design pattern that allows the interface of an existing class to be used as another interface. It is often used to make existing classes work with others without modifying their source code.

**Programmatic Example**

Consider a game where there is a hunter and he hunts lions.

First we have an interface `Lion` that all types of lions have to implement

```C#

interface ILion

{
  void Roar();
}

class AfricanLion : ILion
{
  public void Roar()
  {

  }
}

class AsiaLion : ILion
{
  public void Roar()
  {

  }
}
```

And hunter expects any implementation of `Lion` interface to hunt.

```C#
class Hunter
{
  public void Hunt(ILion lion)
  {

  }
}
```

Now let's say we have to add a `WildDog` in our game so that hunter can hunt that also. But we can't do that directly because dog has a different interface. To make it compatible for our hunter, we will have to create an adapter that is compatible

```C#
// This needs to be added to the game
class WildDog
{
  public void bark()
  {
  }
}

// Adapter around wild dog to make it compatible with our game
class WildDogAdapter : ILion
{
  private WildDog mDog;
  public WildDogAdapter(WildDog dog)
  {
    this.mDog = dog;
  }
  public void Roar()
  {
    mDog.bark();
  }
}
```

And now the `WildDog` can be used in our game using `WildDogAdapter`.

```C#
var wildDog = new WildDog();
var wildDogAdapter = new WildDogAdapter(wildDog);

var hunter = new Hunter();
hunter.Hunt(wildDogAdapter);
```

🚡 Bridge
------
Real world example

> Consider you have a website with different pages and you are supposed to allow the user to change the theme. What would you do? Create multiple copies of each of the pages for each of the themes or would you just create separate theme and load them based on the user's preferences? Bridge pattern allows you to do the second i.e.

![With and without the bridge pattern](https://cloud.githubusercontent.com/assets/11269635/23065293/33b7aea0-f515-11e6-983f-98823c9845ee.png)

In Plain Words

> Bridge pattern is about preferring composition over inheritance. Implementation details are pushed from a hierarchy to another object with a separate hierarchy.

Wikipedia says

> The bridge pattern is a design pattern used in software engineering that is meant to "decouple an abstraction from its implementation so that the two can vary independently"

**Programmatic Example**

Translating our WebPage example from above. Here we have the `WebPage` hierarchy

```C#
interface IWebPage
{
  string GetContent();
}

class About : IWebPage
{
  protected ITheme theme;

  public About(ITheme theme)
  {
    this.theme = theme;
  }

  public string GetContent()
  {
    return $"About page in {theme.GetColor()}";
  }
}

class Careers : IWebPage
{
  protected ITheme theme;

  public Careers(ITheme theme)
  {
    this.theme = theme;
  }

  public string GetContent()
  {
    return $"Careers page in {theme.GetColor()}";
  }
}
```
And the separate theme hierarchy
```C#

interface ITheme
{
  string GetColor();
}

class DarkTheme : ITheme
{
  public string GetColor()
  {
    return "Dark Black";
  }
}

class LightTheme : ITheme
{
  public string GetColor()
  {
    return "Off White";
  }
}

class AquaTheme : ITheme
{
  public string GetColor()
  {
    return "Light blue";
  }
}
```
And both the hierarchies
```C#
var darkTheme = new DarkTheme();
var lightTheme = new LightTheme();

var about= new About(darkTheme);
var careers = new Careers(lightTheme);

Console.WriteLine(about.GetContent()); //Output: About page in Dark Black
Console.WriteLine(careers.GetContent()); //Output: Careers page in Off White
```

## 🌿 Composite

Real world example

> Every organization is composed of employees. Each of the employees has the same features i.e. has a salary, has some responsibilities, may or may not report to someone, may or may not have some subordinates etc.

In plain words

> Composite pattern lets clients treat the individual objects in a uniform manner.

Wikipedia says

> In software engineering, the composite pattern is a partitioning design pattern. The composite pattern describes that a group of objects is to be treated in the same way as a single instance of an object. The intent of a composite is to "compose" objects into tree structures to represent part-whole hierarchies. Implementing the composite pattern lets clients treat individual objects and compositions uniformly.

**Programmatic Example**

Taking our employees example from above. Here we have different employee types

```C#
interface IEmployee
{
  float GetSalary();
  string GetRole();
  string GetName();
}

class Developer : IEmployee
{
  private string mName;
  private float mSalary;
  public Developer(string name, float salary)
  {
    this.mName = name;
    this.mSalary = salary;
  }

  public float GetSalary()
  {
    return this.mSalary;
  }

  public string GetRole()
  {
    return "Developer";
  }

  public string GetName()
  {
    return this.mName;
  }
}

class Designer : IEmployee
{
  private string mName;
  private float mSalary;

  public Designer(string name, float salary)
```

~~~
  {
    this.mName = name;
    this.mSalary = salary;
  }

  public float GetSalary()
  {
    return this.mSalary;
  }

  public string GetRole()
  {
    return "Designer";
  }

  public string GetName()
  {
    return this.mName;
  }
}
```
~~~

Then we have an organization which consists of several different types of employees

~~~
```C#
class Organization
{
  protected List<IEmployee> employees;
~~~

```
  public Organization()
  {
    employees = new List<IEmployee>();
  }

  public void AddEmployee(IEmployee employee)
  {
    employees.Add(employee);
  }

  public float GetNetSalaries()
  {
    float netSalary = 0;

    foreach (var e in employees) {
      netSalary += e.GetSalary();
    }
    return netSalary;
  }
}
```

And then it can be used as

```C#
//Arrange Employees, Organization and add employees
var developer = new Developer("John", 5000);
var designer = new Designer("Arya", 5000);
```

var organization = new Organization();

organization.AddEmployee(developer);

organization.AddEmployee(designer);

Console.WriteLine($"Net Salary of Employees in Organization is {organization.GetNetSalaries():c}");

//Ouptut: Net Salary of Employees in Organization is $10000.00

```

☕ Decorator

-------------

Real world example

> Imagine you run a car service shop offering multiple services. Now how do you calculate the bill to be charged? You pick one service and dynamically keep adding to it the prices for the provided services till you get the final cost. Here each type of service is a decorator.

In plain words

> Decorator pattern lets you dynamically change the behavior of an object at run time by wrapping them in an object of a decorator class.

Wikipedia says

> In object-oriented programming, the decorator pattern is a design pattern that allows behavior to be added to an individual object, either statically or dynamically, without affecting the behavior of other objects from the same class. The decorator pattern is often useful for adhering to the Single Responsibility Principle, as it allows functionality to be divided between classes with unique areas of concern.

**Programmatic Example**

Lets take coffee for example. First of all we have a simple coffee implementing the coffee interface
```

```C#
interface ICoffee
{
  int GetCost();
  string GetDescription();
}

class SimpleCoffee : ICoffee
{
  public int GetCost()
  {
    return 5;
  }

  public string GetDescription()
  {
    return "Simple Coffee";
  }
}
```

We want to make the code extensible to allow options to modify it if required. Lets make some add-ons (decorators)

```C#
class MilkCoffee : ICoffee
{
  private readonly ICoffee mCoffee;

  public MilkCoffee(ICoffee coffee)
```

```
 {
   mCoffee = coffee ?? throw new ArgumentNullException("coffee", "coffee should not be null");
 }
 public int GetCost()
 {
   return mCoffee.GetCost() + 1;
 }

 public string GetDescription()
 {
   return String.Concat(mCoffee.GetDescription(), ", milk");
 }
}

class WhipCoffee : ICoffee
{
 private readonly ICoffee mCoffee;

 public WhipCoffee(ICoffee coffee)
 {
   mCoffee = coffee ?? throw new ArgumentNullException("coffee", "coffee should not be null");
 }
 public int GetCost()
 {
   return mCoffee.GetCost() + 1;
 }

 public string GetDescription()
 {
   return String.Concat(mCoffee.GetDescription(), ", whip");
 }
}

class VanillaCoffee : ICoffee
{
 private readonly ICoffee mCoffee;

 public VanillaCoffee(ICoffee coffee)
 {
   mCoffee = coffee ?? throw new ArgumentNullException("coffee", "coffee should not be null");
 }
 public int GetCost()
 {
   return mCoffee.GetCost() + 1;
 }

 public string GetDescription()
 {
   return String.Concat(mCoffee.GetDescription(), ", vanilla");
 }
}

```

Lets make a coffee now

```C#
var myCoffee = new SimpleCoffee();
```

```
Console.WriteLine($"{myCoffee.GetCost():c}"); // $ 5.00
Console.WriteLine(myCoffee.GetDescription()); // Simple Coffee

var milkCoffee = new MilkCoffee(myCoffee);
Console.WriteLine($"{milkCoffee.GetCost():c}"); // $ 6.00
Console.WriteLine(milkCoffee.GetDescription()); // Simple Coffee, milk

var whipCoffee = new WhipCoffee(milkCoffee);
Console.WriteLine($"{whipCoffee.GetCost():c}"); // $ 7.00
Console.WriteLine(whipCoffee.GetDescription()); // Simple Coffee, milk, whip

var vanillaCoffee = new VanillaCoffee(whipCoffee);
Console.WriteLine($"{vanillaCoffee.GetCost():c}"); // $ 8.00
Console.WriteLine(vanillaCoffee.GetDescription()); // Simple Coffee, milk, whip, vanilla
```

📦 Facade

----------------

Real world example

> How do you turn on the computer? "Hit the power button" you say! That is what you believe because you are using a simple interface that computer provides on the outside, internally it has to do a lot of stuff to make it happen. This simple interface to the complex subsystem is a facade.

In plain words

> Facade pattern provides a simplified interface to a complex subsystem.

Wikipedia says

> A facade is an object that provides a simplified interface to a larger body of code, such as a class library.

**Programmatic Example**

Taking our computer example from above. Here we have the computer class

```C#
class Computer
{
 public void GetElectricShock()
 {
  Console.Write("Ouch!");
 }

 public void MakeSound()
 {
  Console.Write("Beep beep!");
 }

 public void ShowLoadingScreen()
 {
  Console.Write("Loading..");
 }

 public void Bam()
 {
  Console.Write("Ready to be used!");
 }

 public void CloseEverything()
```

```
  {
    Console.Write("Bup bup bup buzzzz!");
  }

  public void Sooth()
  {
    Console.Write("Zzzzz");
  }

  public void PullCurrent()
  {
    Console.Write("Haaah!");
  }
}
```

Here we have the facade

```C#
class ComputerFacade
{
  private readonly Computer mComputer;

  public ComputerFacade(Computer computer)
  {
    this.mComputer = computer ?? throw new ArgumentNullException("computer", "computer cannot be null");
  }

  public void TurnOn()
  {
    mComputer.GetElectricShock();
    mComputer.MakeSound();
    mComputer.ShowLoadingScreen();
    mComputer.Bam();
  }

  public void TurnOff()
  {
    mComputer.CloseEverything();
    mComputer.PullCurrent();
    mComputer.Sooth();
  }
}
```

Now to use the facade

```C#
var computer = new ComputerFacade(new Computer());
computer.TurnOn(); // Ouch! Beep beep! Loading.. Ready to be used!
Console.WriteLine();
computer.TurnOff();  // Bup bup buzzz! Haah! Zzzzz
Console.ReadLine();
```

🍃 Flyweight
---------

Real world example

> Did you ever have fresh tea from some stall? They often make more than one cup that you demanded and save the rest for any other customer so to save the resources e.g. gas etc. Flyweight pattern is all about that i.e. sharing.

In plain words

> It is used to minimize memory usage or computational expenses by sharing as much as possible with similar objects.

Wikipedia says

> In computer programming, flyweight is a software design pattern. A flyweight is an object that minimizes memory use by sharing as much data as possible with other similar objects; it is a way to use objects in large numbers when a simple repeated representation would use an unacceptable amount of memory.

**Programmatic example**

Translating our tea example from above. First of all we have tea types and tea maker

```C#
// Anything that will be cached is flyweight.
// Types of tea here will be flyweights.
class KarakTea
{
}

// Acts as a factory and saves the tea
class TeaMaker
{
  private Dictionary<string,KarakTea> mAvailableTea = new Dictionary<string,KarakTea>();

  public KarakTea Make(string preference)
```

```
 {
   if (!mAvailableTea.ContainsKey(preference))
   {
     mAvailableTea[preference] = new KarakTea();
   }

   return mAvailableTea[preference];
 }
}
```

Then we have the `TeaShop` which takes orders and serves them

```C#
class TeaShop
{
 private Dictionary<int,KarakTea> mOrders = new Dictionary<int,KarakTea>();
 private readonly TeaMaker mTeaMaker;

 public TeaShop(TeaMaker teaMaker)
 {
   mTeaMaker = teaMaker ?? throw new ArgumentNullException("teaMaker", "teaMaker cannot be null");
 }

 public void TakeOrder(string teaType, int table)
 {
   mOrders[table] = mTeaMaker.Make(teaType);
 }
```

```
  public void Serve()
  {
    foreach(var table  in mOrders.Keys){
      Console.WriteLine($"Serving Tea to table # {table}");
    }
  }
}
```

And it can be used as below

```C#
var teaMaker = new TeaMaker();
var teaShop = new TeaShop(teaMaker);

teaShop.TakeOrder("less sugar", 1);
teaShop.TakeOrder("more milk", 2);
teaShop.TakeOrder("without sugar", 5);

teaShop.Serve();
// Serving tea to table# 1
// Serving tea to table# 2
// Serving tea to table# 5
```

🎱 Proxy
-------------------

Real world example

> Have you ever used an access card to go through a door? There are multiple options to open that door i.e. it can be opened either using access card or by pressing a button that bypasses the security. The door's main functionality is to open but there is a proxy added on top of it to add some functionality. Let me better explain it using the code example below.

In plain words

> Using the proxy pattern, a class represents the functionality of another class.

Wikipedia says

> A proxy, in its most general form, is a class functioning as an interface to something else. A proxy is a wrapper or agent object that is being called by the client to access the real serving object behind the scenes. Use of the proxy can simply be forwarding to the real object, or can provide additional logic. In the proxy extra functionality can be provided, for example caching when operations on the real object are resource intensive, or checking preconditions before operations on the real object are invoked.

**Programmatic Example**

Taking our security door example from above. Firstly we have the door interface and an implementation of door

```C#
interface IDoor
{
  void Open();
  void Close();
}

class LabDoor : IDoor
{
  public void Close()
  {
    Console.WriteLine("Closing lab door");
  }

  public void Open()
  {
    Console.WriteLine("Opening lab door");
  }
}
```
Then we have a proxy to secure any doors that we want
```C#
class SecuredDoor : IDoor
{
  private IDoor mDoor;

  public SecuredDoor(IDoor door)
  {
    mDoor = door ?? throw new ArgumentNullException("door", "door can not be null");
  }

  public void Open(string password)
  {
    if (Authenticate(password))
    {
      mDoor.Open();
    }
    else
    {
      Console.WriteLine("Big no! It ain't possible.");
    }
  }


  private bool Authenticate(string password)
  {
    return password == "$ecr@t";
  }


  public void Close()
  {
    mDoor.Close();
  }
}
```

And here is how it can be used

```C#
var door = new SecuredDoor(new LabDoor());
door.Open("invalid"); // Big no! It ain't possible.


door.Open("$ecr@t"); // Opening lab door
door.Close(); // Closing lab door
```

Yet another example would be some sort of data-mapper implementation. For example, I recently made an ODM (Object Data Mapper) for MongoDB using this pattern where I wrote a proxy around mongo classes while utilizing the magic method `__call()`. All the method calls were proxied to the original mongo class and result retrieved was returned as it is but in case of `find` or `findOne` data was mapped to the required class objects and the object was returned instead of `Cursor`.

Behavioral Design Patterns
==========================

In plain words

> It is concerned with assignment of responsibilities between the objects. What makes them different from structural patterns is they don't just specify the structure but also outline the patterns for message passing/communication between them. Or in other words, they assist in answering "How to run a behavior in software component?"

Wikipedia says

> In software engineering, behavioral design patterns are design patterns that identify common communication patterns between objects and realize these patterns. By doing so, these patterns increase flexibility in carrying out this communication.

* [Chain of Responsibility](#-chain-of-responsibility)

* [Command](#-command)

* [Iterator](#-iterator)

* [Mediator](#-mediator)

* [Memento](#-memento)

* [Observer](#-observer)

* [Visitor](#-visitor)

* [Strategy](#-strategy)

* [State](#-state)

* [Template Method](#-template-method)

🔗 Chain of Responsibility

-----------------------

Real world example

> For example, you have three payment methods (`A`, `B` and `C`) setup in your account; each having a different amount in it. `A` has 100 USD, `B` has 300 USD and `C` having 1000 USD and the preference for payments is chosen as `A` then `B` then `C`. You try to purchase something that is worth 210 USD. Using Chain of Responsibility, first of all account `A` will be checked if it can make the purchase, if yes purchase will be made and the chain will be broken. If not, request will move forward to account `B` checking for amount if yes chain will be broken otherwise the request will keep forwarding till it finds the suitable handler. Here `A`, `B` and `C` are links of the chain and the whole phenomenon is Chain of Responsibility.

In plain words

> It helps building a chain of objects. Request enters from one end and keeps going from object to object till it finds the suitable handler.

Wikipedia says

> In object-oriented design, the chain-of-responsibility pattern is a design pattern consisting of a source of command objects and a series of processing objects. Each processing object contains logic that defines the types of command objects that it can handle; the rest are passed to the next processing object in the chain.

**Programmatic Example**

Translating our account example above. First of all we have a base account having the logic for chaining the accounts together and some accounts

```C#
abstract class Account
{
  private Account mSuccessor;
  protected decimal mBalance;

  public void SetNext(Account account)
  {
    mSuccessor = account;
  }

  public void Pay(decimal amountTopay)
  {
    if (CanPay(amountTopay))
    {
      Console.WriteLine($"Paid {amountTopay:c} using {this.GetType().Name}.");
    }
    else if (this.mSuccessor != null)
    {
      Console.WriteLine($"Cannot pay using {this.GetType().Name}. Proceeding..");
      mSuccessor.Pay(amountTopay);
    }
    else
    {
      throw new Exception("None of the accounts have enough balance");
    }
  }
  private bool CanPay(decimal amount)
  {
    return mBalance >= amount;
  }
}

class Bank : Account
{
  public Bank(decimal balance)
  {
    this.mBalance = balance;
  }
}

class Paypal : Account
{
  public Paypal(decimal balance)
  {
    this.mBalance = balance;
  }
}

class Bitcoin : Account
{
  public Bitcoin(decimal balance)
  {
    this.mBalance = balance;
  }
}
```

Now let's prepare the chain using the links defined above (i.e. Bank, Paypal, Bitcoin)

```C#
// Let's prepare a chain like below
//      $bank->$paypal->$bitcoin
//
// First priority bank
//      If bank can't pay then paypal
//      If paypal can't pay then bit coin
var bank = new Bank(100);          // Bank with balance 100
var paypal = new Paypal(200);      // Paypal with balance 200
var bitcoin = new Bitcoin(300);    // Bitcoin with balance 300

bank.SetNext(paypal);
```

```
paypal.SetNext(bitcoin);

// Let's try to pay using the first priority i.e. bank
bank.Pay(259);
// Output will be
// ==============
// Cannot pay using bank. Proceeding ..
// Cannot pay using paypal. Proceeding ..:
// Paid 259 using Bitcoin!
```

👮 Command
-------

Real world example

> A generic example would be you ordering food at a restaurant. You (i.e. `Client`) ask the waiter (i.e. `Invoker`) to bring some food (i.e. `Command`) and waiter simply forwards the request to Chef (i.e. `Receiver`) who has the knowledge of what and how to cook.

> Another example would be you (i.e. `Client`) switching on (i.e. `Command`) the television (i.e. `Receiver`) using a remote control (`Invoker`).

In plain words

> Allows you to encapsulate actions in objects. The key idea behind this pattern is to provide the means to decouple client from receiver.

Wikipedia says

> In object-oriented programming, the command pattern is a behavioral design pattern in which an object is used to encapsulate all information needed to perform an action or trigger an event at a later time. This information includes the method name, the object that owns the method and values for the method parameters.

**Programmatic Example**

First of all we have the receiver that has the implementation of every action that could be performed

```C#
// Receiver
class Bulb
{
  public void TurnOn()
  {
    Console.WriteLine("Bulb has been lit");
  }

  public void TurnOff()
  {
    Console.WriteLine("Darkness!");
  }
}
```

then we have an interface that each of the commands are going to implement and then we have a set of commands

```C#
interface ICommand
{
  void Execute();
  void Undo();
  void Redo();
}

// Command
```

```
class TurnOn : ICommand
{
  private Bulb mBulb;

  public TurnOn(Bulb bulb)
  {
    mBulb = bulb ?? throw new ArgumentNullException("Bulb", "Bulb cannot be null");
  }

  public void Execute()
  {
    mBulb.TurnOn();
  }

  public void Undo()
  {
    mBulb.TurnOff();
  }

  public void Redo()
  {
    Execute();
  }
}

class TurnOff : ICommand
{
  private Bulb mBulb;
  public TurnOff(Bulb bulb)
  {
    mBulb = bulb ?? throw new ArgumentNullException("Bulb", "Bulb cannot be null");
  }

  public void Execute()
  {
    mBulb.TurnOff();
  }

  public void Undo()
  {
    mBulb.TurnOn();
  }

  public void Redo()
  {
    Execute();
  }
}
```
Then we have an `Invoker` with whom the client will interact to process any commands
```C#
// Invoker
class RemoteControl
{
  public void Submit(ICommand command)
  {
    command.Execute();
  }
}
```

Finally let's see how we can use it in our client

```C#
  var bulb = new Bulb();


  var turnOn = new TurnOn(bulb);
  var turnOff = new TurnOff(bulb);


  var remote = new RemoteControl();
  remote.Submit(turnOn); // Bulb has been lit!
  remote.Submit(turnOff); // Darkness!


  Console.ReadLine();
```

Command pattern can also be used to implement a transaction based system. Where you keep maintaining the history of commands as soon as you execute them. If the final command is successfully executed, all good otherwise just iterate through the history and keep executing the `undo` on all the executed commands.

➿ Iterator

--------

Real world example

> An old radio set will be a good example of iterator, where user could start at some channel and then use next or previous buttons to go through the respective channels. Or take an example of MP3 player or a TV set where you could press the next and previous buttons to go through the consecutive channels or in other words they all provide an interface to iterate through the respective channels, songs or radio stations.

In plain words

> It presents a way to access the elements of an object without exposing the underlying presentation.

Wikipedia says

> In object-oriented programming, the iterator pattern is a design pattern in which an iterator is used to traverse a container and access the container's elements. The iterator pattern decouples algorithms from containers; in some cases, algorithms are necessarily container-specific and thus cannot be decoupled.

**Programmatic example**

In C# it can be done by implementing IEnumerable<T>. Translating our radio statiIons example from above. First of all we have `RadioStation`

```C#
class RadioStation
{
  private float mFrequency;

  public RadioStation(float frequency)
  {
    mFrequency = frequency;
  }

  public float GetFrequecy()
  {
    return mFrequency;
  }

}
```

Then we have our iterator

```C#
class StationList : IEnumerable<RadioStation>
{
  List<RadioStation> mStations = new List<RadioStation>();

  public RadioStation this[int index]
  {
    get { return mStations[index]; }
    set { mStations.Insert(index, value); }
  }

  public void Add(RadioStation station)
  {
    mStations.Add(station);
  }

  public void Remove(RadioStation station)
  {
    mStations.Remove(station);
  }

  public IEnumerator<RadioStation> GetEnumerator()
  {
    return this.GetEnumerator();
  }
  IEnumerator IEnumerable.GetEnumerator()
  {
    //Use can switch to this internal collection if you do not want to transform
    //return mStations.GetEnumerator();

    //use this if you want to transform the object before rendering
    foreach (var x in mStations)
    {
      yield return x;
    }
  }
}
```
And then it can be used as
```C#
var stations = new StationList();
var station1 = new RadioStation(89);
stations.Add(station1);

var station2 = new RadioStation(101);
stations.Add(station2);

var station3 = new RadioStation(102);
stations.Add(station3);

foreach(var x in stations)
{
  Console.Write(x.GetFrequecy());
}


var q = stations.Where(x => x.GetFrequecy() == 89).FirstOrDefault();

Console.WriteLine(q.GetFrequecy());


Console.ReadLine();
```

👽 Mediator

========

Real world example

> A general example would be when you talk to someone on your mobile phone, there is a network provider sitting between you and them and your conversation goes through it instead of being directly sent. In this case network provider is mediator.

In plain words

> Mediator pattern adds a third party object (called mediator) to control the interaction between two objects (called colleagues). It helps reduce the coupling between the classes communicating with each other. Because now they don't need to have the knowledge of each other's implementation.

Wikipedia says

> In software engineering, the mediator pattern defines an object that encapsulates how a set of objects interact. This pattern is considered to be a behavioral pattern due to the way it can alter the program's running behavior.

**Programmatic Example**

Here is the simplest example of a chat room (i.e. mediator) with users (i.e. colleagues) sending messages to each other.

First of all, we have the mediator i.e. the chat room

```C#
interface IChatRoomMediator
{
  void ShowMessage(User user, string message);
}

//Mediator
class ChatRoom : IChatRoomMediator
{
  public void ShowMessage(User user, string message)
  {
    Console.WriteLine($"{DateTime.Now.ToString("MMMM dd, H:mm")} [{user.GetName()}]:{message}");
  }
}
```

Then we have our users i.e. colleagues

```C#
class User
{
  private string mName;
  private IChatRoomMediator mChatRoom;

  public User(string name, IChatRoomMediator chatroom)
  {
    mChatRoom = chatroom;
    mName = name;
  }

  public string GetName()
  {
    return mName;
  }

  public void Send(string message)
  {
    mChatRoom.ShowMessage(this, message);
  }
}
```

And the usage

```C#
var mediator = new ChatRoom();

var john = new User("John", mediator);
var jane = new User("Jane", mediator);

john.Send("Hi there!");
jane.Send("Hey!");

//April 14, 20:05[John]:Hi there!
//April 14, 20:05[Jane]:Hey!
```

💾 Memento
-------

Real world example

> Take the example of calculator (i.e. originator), where whenever you perform some calculation the last calculation is saved in memory (i.e. memento) so that you can get back to it and maybe get it restored using some action buttons (i.e. caretaker).

In plain words

> Memento pattern is about capturing and storing the current state of an object in a manner that it can be restored later on in a smooth manner.

Wikipedia says

> The memento pattern is a software design pattern that provides the ability to restore an object to its previous state (undo via rollback).

Usually useful when you need to provide some sort of undo functionality.

**Programmatic Example**

Lets take an example of text editor which keeps saving the state from time to time and that you can restore if you want.

First of all we have our memento object that will be able to hold the editor state

```C#
class EditorMemento
{
  private string mContent;

  public EditorMemento(string content)
  {
    mContent = content;
  }

  public string Content
  {
    get
    {
      return mContent;
    }
  }
}
```

Then we have our editor i.e. originator that is going to use memento object

```C#
class Editor {

  private string mContent = string.Empty;
  private EditorMemento memento;

  public Editor()
  {
    memento = new EditorMemento(string.Empty);
  }

  public void Type(string words)
  {
    mContent = String.Concat(mContent," ", words);
  }
```

```
 public string Content
 {
  get
  {
   return mContent;
  }
 }

 public void Save()
 {
  memento = new EditorMemento(mContent);
 }

 public void Restore()
 {
  mContent = memento.Content;
 }
}
```

And then it can be used as

```C#
var editor = new Editor();

//Type some stuff
editor.Type("This is the first sentence.");
editor.Type("This is second.");
```

// Save the state to restore to : This is the first sentence. This is second.

editor.Save();

//Type some more

editor.Type("This is third.");

//Output the content

Console.WriteLine(editor.Content); // This is the first sentence. This is second. This is third.

//Restoring to last saved state

editor.Restore();

Console.Write(editor.Content); // This is the first sentence. This is second

```

😎 Observer

--------

Real world example

> A good example would be the job seekers where they subscribe to some job posting site and they are notified whenever there is a matching job opportunity.

In plain words

> Defines a dependency between objects so that whenever an object changes its state, all its dependents are notified.

Wikipedia says
```

> The observer pattern is a software design pattern in which an object, called the subject, maintains a list of its dependents, called observers, and notifies them automatically of any state changes, usually by calling one of their methods.

**Programmatic example**

Translating our example from above. First of all we have job seekers that need to be notified for a job posting

```C#
class JobPost
{
  public string Title { get; private set; }

  public JobPost(string title)
  {
    Title = title;
  }
}
class JobSeeker : IObserver<JobPost>
{
  public string Name { get; private set; }

  public JobSeeker(string name)
  {
    Name = name;
  }

  //Method is not being called by JobPostings class currently
  public void OnCompleted()
  {
    //No Implementation
  }

  //Method is not being called by JobPostings class currently
  public void OnError(Exception error)
  {
    //No Implementation
  }

  public void OnNext(JobPost value)
  {
    Console.WriteLine($"Hi {Name} ! New job posted: {value.Title}");
  }
}
```

Then we have our job postings to which the job seekers will subscribe

```C#
class JobPostings : IObservable<JobPost>
{
  private List<IObserver<JobPost>> mObservers;
  private List<JobPost> mJobPostings;

  public JobPostings()
  {
    mObservers = new List<IObserver<JobPost>>();
    mJobPostings = new List<JobPost>();
  }

  public IDisposable Subscribe(IObserver<JobPost> observer)
  {
    // Check whether observer is already registered. If not, add it
    if (!mObservers.Contains(observer))
    {
      mObservers.Add(observer);
    }
    return new Unsubscriber<JobPost>(mObservers, observer);
  }

  private void Notify(JobPost jobPost)
  {
    foreach(var observer in mObservers)
    {
      observer.OnNext(jobPost);
    }
  }

  public void AddJob(JobPost jobPost)
  {
    mJobPostings.Add(jobPost);
    Notify(jobPost);
  }

}

internal class Unsubscriber<JobPost> : IDisposable
{
  private List<IObserver<JobPost>> mObservers;
  private IObserver<JobPost> mObserver;
  internal Unsubscriber(List<IObserver<JobPost>> observers, IObserver<JobPost> observer)
  {
    this.mObservers = observers;
    this.mObserver = observer;
  }

  public void Dispose()
  {
    if (mObservers.Contains(mObserver))
      mObservers.Remove(mObserver);
  }
}
```
Then it can be used as
```C#
//Create Subscribers
var johnDoe = new JobSeeker("John Doe");
var janeDoe = new JobSeeker("Jane Doe");

//Create publisher and attch subscribers
var jobPostings = new JobPostings();
jobPostings.Subscribe(johnDoe);
jobPostings.Subscribe(janeDoe);

//Add a new job and see if subscribers get notified
jobPostings.AddJob(new JobPost("Software Engineer"));

//Output
```

```
// Hi John Doe! New job posted: Software Engineer

// Hi Jane Doe! New job posted: Software Engineer


Console.ReadLine();
```

🏃 Visitor

-------

Real world example

> Consider someone visiting Dubai. They just need a way (i.e. visa) to enter Dubai. After arrival, they can come and visit any place in Dubai on their own without having to ask for permission or to do some leg work in order to visit any place here; just let them know of a place and they can visit it. Visitor pattern lets you do just that, it helps you add places to visit so that they can visit as much as they can without having to do any legwork.

In plain words

> Visitor pattern lets you add further operations to objects without having to modify them.

Wikipedia says

> In object-oriented programming and software engineering, the visitor design pattern is a way of separating an algorithm from an object structure on which it operates. A practical result of this separation is the ability to add new operations to existing object structures without modifying those structures. It is one way to follow the open/closed principle.

**Programmatic example**

Let's take an example of a zoo simulation where we have several different kinds of animals and we have to make them Sound. Let's translate this using visitor pattern

```C#
// Visitee
```

```
interface IAnimal
{
  void Accept(IAnimalOperation operation);
}

// Visitor
interface IAnimalOperation
{
  void VisitMonkey(Monkey monkey);
  void VisitLion(Lion lion);
  void VisitDolphin(Dolphin dolphin);
}
```
Then we have our implementations for the animals
```C#
class Monkey : IAnimal
{
  public void Shout()
  {
    Console.WriteLine("Oooh o aa aa!");
  }

  public void Accept(IAnimalOperation operation)
  {
      operation.VisitMonkey(this);
  }
}

class Lion : IAnimal
```

```
{
  public void Roar()
  {
    Console.WriteLine("Roaar!");
  }

  public void Accept(IAnimalOperation operation)
  {
      operation.VisitLion(this);
  }
}

class Dolphin : IAnimal
{
  public void Speak()
  {
    Console.WriteLine("Tuut tittu tuutt!");
  }

  public void Accept(IAnimalOperation operation)
  {
      operation.VisitDolphin(this);
  }
}
```
Let's implement our visitor
```C#
class Speak : IAnimalOperation
{
  public void VisitDolphin(Dolphin dolphin)
  {
    dolphin.Speak();
  }

  public void VisitLion(Lion lion)
  {
    lion.Roar();
  }

  public void VisitMonkey(Monkey monkey)
  {
    monkey.Shout();
  }
}
```

And then it can be used as

```C#
var monkey = new Monkey();
var lion = new Lion();
var dolphin = new Dolphin();

var speak = new Speak();

monkey.Accept(speak);    // Ooh oo aa aa!
lion.Accept(speak);      // Roaaar!
dolphin.Accept(speak);   // Tuut tutt tuutt!
```

We could have done this simply by having an inheritance hierarchy for the animals but then we would have to modify the animals whenever we would have to add new actions to animals. But now we will not have to change them. For example, let's say we are asked to add the jump behavior to the animals, we can simply add that by creating a new visitor i.e.

```C#
class Jump : IAnimalOperation
{
  public void VisitDolphin(Dolphin dolphin)
  {
    Console.WriteLine("Walked on water a little and disappeared!");
  }

  public void VisitLion(Lion lion)
  {
    Console.WriteLine("Jumped 7 feet! Back on the ground!");
  }

  public void VisitMonkey(Monkey monkey)
  {
    Console.WriteLine("Jumped 20 feet high! on to the tree!");
  }
}
```

And for the usage

```C#
var jump = new Jump();

monkey.Accept(speak);   // Ooh oo aa aa!
monkey.Accept(jump);    // Jumped 20 feet high! on to the tree!

lion.Accept(speak);     // Roaaar!
lion.Accept(jump);      // Jumped 7 feet! Back on the ground!

dolphin.Accept(speak);  // Tuut tutt tuutt!
dolphin.Accept(jump);   // Walked on water a little and disappeared
```

💡 Strategy

--------

Real world example

> Consider the example of sorting, we implemented bubble sort but the data started to grow and bubble sort started getting very slow. In order to tackle this we implemented Quick sort. But now although the quick sort algorithm was doing better for large datasets, it was very slow for smaller datasets. In order to handle this we implemented a strategy where for small datasets, bubble sort will be used and for larger, quick sort.

In plain words

> Strategy pattern allows you to switch the algorithm or strategy based upon the situation.

Wikipedia says

> In computer programming, the strategy pattern (also known as the policy pattern) is a behavioural software design pattern that enables an algorithm's behavior to be selected at runtime.

**Programmatic example**

Translating our example from above. First of all we have our strategy interface and different strategy implementations

```C#
interface ISortStrategy
{
  List<int> Sort(List<int> dataset);
}

class BubbleSortStrategy : ISortStrategy
{
  public List<int> Sort(List<int> dataset)
  {
    Console.WriteLine("Sorting using Bubble Sort !");
    return dataset;
  }
}

class QuickSortStrategy : ISortStrategy
{
  public List<int> Sort(List<int> dataset)
  {
    Console.WriteLine("Sorting using Quick Sort !");
    return dataset;
  }
}
```

And then we have our client that is going to use any strategy

```C#
class Sorter
```

```
{
  private readonly ISortStrategy mSorter;

  public Sorter(ISortStrategy sorter)
  {
    mSorter = sorter;
  }

  public List<int> Sort(List<int> unSortedList)
  {
    return mSorter.Sort(unSortedList);
  }
}
```
And it can be used as
```C#
var unSortedList = new List<int> { 1, 10, 2, 16, 19 };

var sorter = new Sorter(new BubbleSortStrategy());
sorter.Sort(unSortedList); // // Output : Sorting using Bubble Sort !

sorter = new Sorter(new QuickSortStrategy());
sorter.Sort(unSortedList); // // Output : Sorting using Quick Sort !
```

💢 State
-----
Real world example

> Imagine you are using some drawing application, you choose the paint brush to draw. Now the brush changes its behavior based on the selected color i.e. if you have chosen red color it will draw in red, if blue then it will be in blue etc.

In plain words

> It lets you change the behavior of a class when the state changes.

Wikipedia says

> The state pattern is a behavioral software design pattern that implements a state machine in an object-oriented way. With the state pattern, a state machine is implemented by implementing each individual state as a derived class of the state pattern interface, and implementing state transitions by invoking methods defined by the pattern's superclass.

> The state pattern can be interpreted as a strategy pattern which is able to switch the current strategy through invocations of methods defined in the pattern's interface.

**Programmatic example**

Let's take an example of text editor, it lets you change the state of text that is typed i.e. if you have selected bold, it starts writing in bold, if italic then in italics etc.

First of all we have our state interface and some state implementations

```C#
interface IWritingState {

  void Write(string words);

}

class UpperCase : IWritingState
{
  public void Write(string words)
  {
    Console.WriteLine(words.ToUpper());
  }
}

class LowerCase : IWritingState
{
  public void Write(string words)
  {
    Console.WriteLine(words.ToLower());
  }
}

class DefaultText : IWritingState
{
  public void Write(string words)
  {
    Console.WriteLine(words);
  }
}
```
Then we have our editor
```C#
class TextEditor {

  private IWritingState mState;

  public TextEditor()
```

```
 {
  mState = new DefaultText();
 }

 public void SetState(IWritingState state)
 {
  mState = state;
 }

 public void Type(string words)
 {
  mState.Write(words);
 }

}
```
And then it can be used as
```C#
var editor = new TextEditor();

editor.Type("First line");

editor.SetState(new UpperCase());

editor.Type("Second Line");
editor.Type("Third Line");

editor.SetState(new LowerCase());
editor.Type("Fourth Line");
editor.Type("Fifthe Line");

// Output:
// First line
// SECOND LINE
// THIRD LINE
// fourth line
// fifth line
```

📒 Template Method

---------------

Real world example

> Suppose we are getting some house built. The steps for building might look like
>
> - Prepare the base of house
> - Build the walls
> - Add roof
> - Add other floors

> The order of these steps could never be changed i.e. you can't build the roof before building the walls etc but each of the steps could be modified for example walls can be made of wood or polyester or stone.

In plain words

> Template method defines the skeleton of how a certain algorithm could be performed, but defers the implementation of those steps to the children classes.

Wikipedia says

> In software engineering, the template method pattern is a behavioral design pattern that defines the program skeleton of an algorithm in an operation, deferring some steps to subclasses. It lets one redefine certain steps of an algorithm without changing the algorithm's structure.

**Programmatic Example**

Imagine we have a build tool that helps us test, lint, build, generate build reports (i.e. code coverage reports, linting report etc) and deploy our app on the test server.

First of all we have our base class that specifies the skeleton for the build algorithm

```C#
abstract class Builder
{
  // Template method
  public void Build()
  {
    Test();
    Lint();
    Assemble();
    Deploy();
  }

  abstract public void Test();
  abstract public void Lint();
  abstract public void Assemble();
  abstract public void Deploy();
}
```

Then we can have our implementations

```C#
class AndroidBuilder : Builder
{
 public override void Assemble()
 {
  Console.WriteLine("Assembling the android build");
 }

 public override void Deploy()
 {
  Console.WriteLine("Deploying android build to server");
 }

 public override void Lint()
 {
  Console.WriteLine("Linting the android code");
 }

 public override void Test()
 {
  Console.WriteLine("Running android tests");
 }
}


class IosBuilder : Builder
{
 public override void Assemble()
```

```
  {
    Console.WriteLine("Assembling the ios build");
  }

  public override void Deploy()
  {
    Console.WriteLine("Deploying ios build to server");
  }

  public override void Lint()
  {
    Console.WriteLine("Linting the ios code");
  }

  public override void Test()
  {
    Console.WriteLine("Running ios tests");
  }
}

```
And then it can be used as

```C#
var androidBuilder = new AndroidBuilder();
androidBuilder.Build();

// Output:
// Running android tests
```

// Linting the android code

// Assembling the android build

// Deploying android build to server

var iosBuilder = new IosBuilder();

iosBuilder.Build();

// Output:

// Running ios tests

// Linting the ios code

// Assembling the ios build

// Deploying ios build to server
```

## 🚦 Wrap Up Folks

And that about wraps it up. I will continue to improve this, so you might want to watch/star this repository to revisit. Also, I have plans on writing the same about the architectural patterns, stay tuned for it.

## 👫 Contribution

- Report issues

- Open pull request with improvements

- Spread the word

- Contact me on <a href="https://twitter.com/anupavanm">Twitter</a>

## License
```