

Essential Design Patterns in C#

by Steven Nikolic

ABOUT THE AUTHOR AND THESE LESSONS

Programming is a skill that, like any, is best learned through hands-on practice and by filling in underlying theory as you go. I learned programming the way that most seem to: guided by natural interest and moving at my own pace (ie: self-taught). Whether I wanted to develop my own website, or create a video game, I always enjoyed the challenge of diving in, learning something new, and building something unique. Each project was a massive learning opportunity, and after a decade and many projects later, I'm happy to say that I'm still learning.

I didn't seek out to learn design patterns when I first started out. That early stage was for getting comfortable with the basics common to all programming pursuits. Realistically, design patterns won't make a lot of sense until you've at least tackled one project that is somewhat large in scope. Only then, in retrospect, can you fully appreciate the paradoxical simplicity you gain by constructing more elaborate design schemes. In short, you will have learned the hard way how quickly code can get out of control, difficult to keep track of and prone to maddening error. Only then can you appreciate how, and why, structurally clean code is so important.

With some messy projects under my belt, I began looking up design patterns whenever I came across a structural problem that could affect the broader system being created. For example, if I was building a product inventory that had unknowable properties in the future, I would look up the patterns that best represented a general solution to that problem. I would then use the established pattern as a guide, and essentially create a version of it unique to the system I was in charge of designing.

Eventually I had familiarized myself with many design patterns that suited different use cases in a wide variety of domains. Some patterns were used often, in almost all projects, while others were more specific but nonetheless powerful and worth knowing. I didn't find it so important to remember any of the patterns deeply so as to be able to write from scratch without reference. Instead, the important part was just being aware that the patterns exist, and broadly what problem they solve.

Since I generally knew how to use design patterns, the dirty work usually was in finding suitable (easy to read) reference material in order to work quickly and effectively. Many examples I found were overly complicated, wordy or even in an unfamiliar language that required some degree of mental conversion.

I don't believe programming languages are worth fussing about. What is important is identifying programmatic form - the underlying functionality common to virtually all programming languages. Design patterns are best understood as representing this common form. Concepts that you can port to any language or problem set, and be a good starting point for finding a solution.

Still, a language must be chosen. Much of my work, partially out of chance and circumstance, has led me to predominantly work in C#. As such, it's natural for me to construct this compendium of design patterns, that were personally useful to me, in that language.

Java and C# bear striking similarity, and neither should be terribly unfamiliar to those working somewhere within the C-like paradigm. So I think my choice of language here is as good as any to represent these useful patterns. Hopefully that choice makes your learning of these patterns easier for similar reasons. However if you work in other languages I hope you still find the simple approach I took here useful just the same. If you do find any of this useful, a small [donation](#) is greatly appreciated.

- Steven Nikolic

Table of Contents

[Introduction](#)

[Singleton](#)

[Prototype](#)

[Factory Method](#)

[Abstract Factory](#)

[Structural Wrappers](#)

[Adapter](#)

[Bridge](#)

[Proxy](#)

[Facade](#)

[Decorator](#)

[Composite](#)

[Template Method](#)

[State](#)

[Strategy](#)

[Observer](#)

[Command](#)

[Iterator](#)

[CheatSheet](#)

1. Design Patterns – Introduction

WHAT ARE DESIGN PATTERNS?

In the following lessons we're going to cover the topic of design patterns applied using the language of C#. All of the patterns covered originate from the seminal book titled "**Design Patterns: Elements of Reusable Object-Oriented Software.**" It was first published in 1994 by what have been known colloquially as the "**Gang of Four**" (GoF). These four authors - *Erich Gamma, John Vlissides, Ralph Johnson and Richard Helm* - identified and cataloged in this book the most commonly occurring problems in need of coding solutions in software development.

To be specific, they recognized coding tasks that frequently pop up when developing software, and had set out to establish robust and clear solutions so that you, the developer, needn't reinvent the wheel. By "tasks" we're referring to basic objectives you would seek out to accomplish when writing code. So, for example, if you have an object in code that has certain base characteristics and yet will have an indefinite amount of components added to it (sort of like adding toppings on a pizza or features on a new car) you can reach out and grab the "Decorator" design pattern and use that as a solid starting point.

Being a "pattern" what you get is a sort of scaffold or templated starting point. Fitting your real-world code into an established pattern can take many forms, including mixing, merging, and partially using other patterns, or using simplified (or more complex) versions of them, and so forth.

In that way, design patterns aren't monolithic but rather can be flexible and used as you see fit to help boost the performance, organization and durability of the code that you write. And simply knowing, writing and identifying design patterns can be a great way to take you to that "next level" in your software development career.

WHEN SHOULD I LEARN THESE PATTERNS?

You might be asking yourself if you're at that stage where knowing design patterns is that next step you should take. The general rule for answering this concern is this: once you've solidified the foundations of writing code, grasping the basics of OOP and having written some real-world production code, you'll likely be ready to appreciate the nuance and utility of having these design patterns at arms reach.

Beginners, by definition, don't fully know what they don't know. They'll apply what they've learned without a broader plan or sense of pitfalls that can spring up. They have yet to work through the trouble of writing spaghetti code and broken systems in need of rewrites before they can fully appreciate (and understand) how applying something like design patterns can be of use. At this "growing pains" stage in your software development journey, you'll want to avoid making the same mistakes repeatedly, and should look towards ways of simplifying the job of solving problems at hand rather than worrying endlessly about how the code is setup to begin with.

To that end, understanding design patterns just becomes another tool - albeit a powerful one - in your programming toolkit. As with any tool, *you don't always have to take out the tool* and use it (although it's common for those who've just learned how to use the tool to use it as much as possible!). However, simply *knowing the tool is available* will make you better at identifying a possible solution to a well-known problem.

HOW YOU WILL END UP USING THESE PATTERNS

You won't necessarily - nor would you be expected to - have a photographic memory of how these patterns are constructed. But rather having worked through them in these lessons, you'll come to understand *how* they are constructed, *what* problems they solve and *where* they are best used so that you can refer back to what you learned and implement them later.

So, for example, when you're at a coding cross-roads in the future you might come across a situation where you want to limit instantiation of objects and you'll think:

"That sounds like the *Singleton Pattern*. I'll refresh my memory on how to implement that."

Or, perhaps, you're writing code that accesses some 3rd party plugin code that you know might change or become replaced so you'll say to yourself:

"I'll just write a wrapper class that encapsulates 3rd party objects within a *Bridge Pattern*."

Whether or not you fully implement any particular pattern in your development career, simply *understanding them* should make your code become more robust and resilient to change. That's because design patterns typically insist on implementing tried-and-true methods of code separation and reuse; insisting on a finer understanding of OOP cornerstones such as Interfaces, Abstraction, Inheritance and Composition.

You will also likely become better at reading other people's code. Design patterns are used everywhere, but if you've never seen them your mind will likely boggle at what *exactly* it is that you are looking at. A lot of complicated code holds much of its complexity through the work of a particular design pattern, or else holds a partial resemblance to one. Being able to identify and catalogue a coding pattern - through commonly used naming schemes and overall code structure, whether or not it was created intentionally or with grace - often goes a long way towards answering the perennial coding question of "what exactly am I looking at?" With this knowledge at your disposal, you can free up your mind to abstract away a lot more code, tabling concerns about *form* and allowing you to better focus on the *guts* of the problem.

So, if you've never come across design patterns up until now, what should be clear is that design patterns are blueprint-type solutions to common programming problems. They are high level patterns which differ from architectural patterns such as MVC and other more organizational or principle-based concepts and patterns. They aren't a silver-bullet to all coding problems and do not need to be implemented wherever possible. Because they add a level of complexity to your code, you always have to ask if that extra complexity will provide a return on time investment. Nonetheless, familiarizing yourself with these patterns pays dividends whether you use them often, or only selectively.

FAST-TRACKING WITH THESE LESSONS

After hearing all of this you may think "*Why don't I just pick up the Gang of Four book on design patterns?*"

You certainly *could* do that. The basic foundation for design patterns, as well as basic implementation, hasn't changed much since the GoF book was written. However, chances are that you want to fast-track your learning, see some concrete usage examples, and all in a language you are familiar with - like C#.

The original GoF book was predominantly written in the **SmallTalk** programming language. And although the basic OOP principles of SmallTalk and its syntax bears resemble to other OOP languages, it's just *that much easier* to stick with the programming language you want to apply the principles to and are most comfortable with.

Another benefit to learning these lessons is the order in which you learn them. You'll be taking a step-by-step journey through the most-often-used and most interesting patterns, starting with simpler ones and then moving on to more complex examples. And in many cases, certain patterns are better understood having already looked at similar ones.

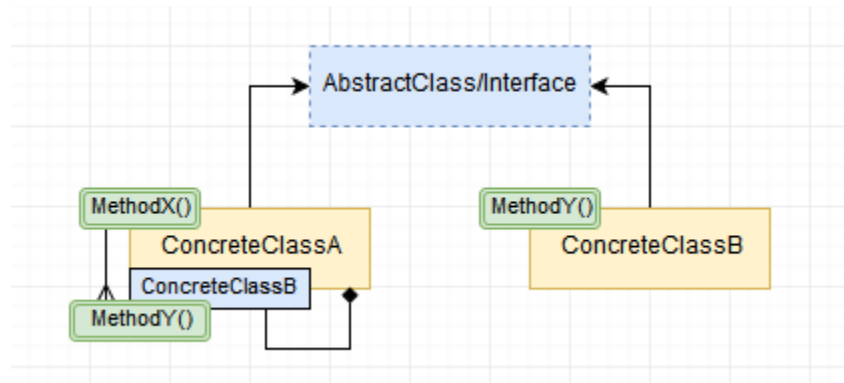
We'll look at the sub-category that each pattern is a part of, whether:

- **Structural** (aimed at keeping code robust, clear and resilient to change)
- **Behavioural** (aimed at completing core tasks by focusing on methods)
- **Creational** (aimed at the efficient creation of new objects).

All important bases will be covered using simple, yet meaningful, examples that give you a better understanding of **why** a particular pattern exists and **what** problem it solves. You'll also have reference materials such as simple non-UML diagrams, synopses and memory-helper phrases so as to make it easier to identify the patterns in the wild and refresh your understanding.

Through simple *scaffold* code and more in-depth *real-world* examples, you'll learn **how** the patterns operate as you're constructing them. Other materials often just throw a completed pattern on the screen - like you might see in a book - and expect you to understand it in full. By contrast, our approach will be to step through the line of thinking at every point so as to clarify how everything is connected to form the complete pattern. At the end of every lesson we'll also have a wrap-up and short quiz to help solidify your understanding of each pattern. The aim is to keep the lessons short, intensive and hands-on.

UNDERSTANDING THE PATTERN DIAGRAMS USED



Most pattern diagrams used in the lessons follow these basic guidelines, or are otherwise self-explanatory.

- Blue-shaded blocks with dotted lines denote **abstract** class or **interface** definitions.
- Yellow-shaded blocks denote **non-abstract** (IE: “concrete”) class definitions.
- Green-shaded blocks denote **method** definitions, and which classes/instances they belong to.
- Blue-shaded blocks that connect to class definitions denote **object instances** or a class-level variable.
- **Method calls** are denoted by a line that ends in three-prongs. With the empty end denoting the caller, and the three-prong end denoting the method being called.
- **Inheritance** is denoted with arrows, with the empty end denoting the inheriting child class, and the arrow end denoting the parent class being inherited from.

The diagram example above can be read as follows:

- **ConcreteClassA** and **ConcreteClassB** inherits from **AbstractClass/Interface**
- **ConcreteClassA** contains an instance of **ConcreteClassB**
- **ConcreteClassA.MethodX()** calls **ConcreteClassB.MethodY()** via the **ConcreteClassB** instance

One last note before we move on to the first lesson. I chose 16 design patterns originating from the GoF book that I thought were the most instructive, frequently used and important to know. There are certainly a lot of other patterns in the wild. Some have more niche usage, while others are more broad in definition and scope, or border on design *principle* more than a particular implementation *pattern* (for instance: **Dependency Injection**). With that out of the way, let's go right into our first lesson on the deceptively simple, and ubiquitous, **Singleton** pattern.

2. Singleton Pattern

PATTERN TYPE: Creational

FREQUENCY OF USE: High

MEMORY HELPER: Unique Object

PATTERN OBJECTIVE: Limit instantiation of a given class to a single instance, ensuring the uniqueness of the object in the system. Also, providing a global access point to the static instance.

MAIN REFERENCE: <https://msdn.microsoft.com/en-us/library/ff650316.aspx>

PREAMBLE: The most common, and probably the simplest pattern you will come across in software development, is the Singleton pattern.

If you don't know anything about design patterns the Singleton Pattern is a great place to start as it's fairly easy to understand and its usefulness is straightforward and clear.

Design patterns exist to solve particular, and common, problems faced when coding. The main problem that a Singleton solves is *it limits the instantiation of a particular class (which is built using the pattern) to a single object.*

This is particularly useful when you know ahead of time that you will only need a *single instance* of a given object, and especially useful when having multiple instances of that object (however unlikely that is to happen) could cause a real problem in the system.

Let's consider some hypothetical examples. Say you have an application that has to make a single network connection. You want to make sure you're using that same connection rather than creating a new one every time you want to make/use that connection. Or perhaps you have an app that's making use of the underlying file system.

In both of these examples, the object that you will depend on for that service shouldn't change. You're not going to need new connections to a network or to a new file system. To prevent any possible problems where multiple instances could be created in these cases, simply make the object that handles access to those resources a **Singleton** to strictly enforce that rule in your code.

BASIC EXAMPLE OF SINGLETON IN USE

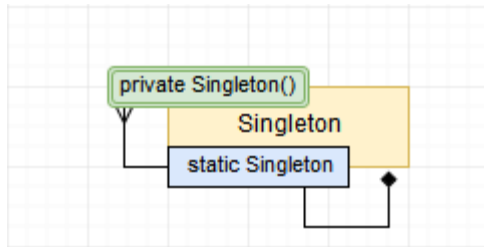
With this in mind, let's look at the "scaffolding" structure of a basic Singleton class:

```
class Singleton
{
    //Constructor
    private Singleton() { }

    private static Singleton _instance;
    public static Singleton Instance
    {
        get
        {
            if (_instance == null)
                _instance = new Singleton();

            return _instance;
        }
    }
}
```

You can visualize the important part of this structure with the following diagram:



This looks deceptively simple, but if you look at it carefully you'll see there are some oddities that you may not be familiar with. First, we see that the class has a private constructor.

```
private Singleton() { }
```

This is a bit odd looking but it doesn't violate any C# rule. You've probably just gotten used to the idea of constructors being necessarily public because you always hand off instance creation of the object to some outside class.

Well, when creating a Singleton, you explicitly want to forbid this possibility. You *do* want an instance, but *do not* want to let other classes create the instance, because then you're depending on the outside class to contain the logic to limit the instance to exactly one.

But you can't really do that. Every class, in principle, will have the right to create an instance. So, you absolutely *have* to limit instance creation by containing it in the class itself that is intended to be a Singleton.

That brings us to the next part of making sense of this peculiarity. We see that the Singleton contains an *instance of itself*.

```
private static Singleton _instance;
```

Again, on the surface this might appear strange because you're used to separating the idea of a class as a blueprint and the object as just one possible instance of that blueprint. Basically what we're seeing here is a class blueprint which really is used to make *just one* version of itself and only one time. Again, odd, but not logically unsound or violating any rules, and clearly enforces the rule of keeping instantiation to a single object.

So now we turn to the final piece of the puzzle which is the actual creation of the instance. We see a *getter* being used to reference the instance:

```
public static Singleton Instance
{
    get
    {
        if (_instance == null)
            _instance = new Singleton();

        return _instance;
    }
}
```

Basically, what this does is It *creates* the instance, and *returns* it, whenever (and wherever) we reference the Instance property for the first time. Every other time after that, it *just returns* the instance.

This “create on first-time reference” is ensured by the null check:

```
if (_instance == null)
```

To make this a bit more clear how this is all working, let's reference the Instance in another class:

```
class Program
{
    static void Main()
    {
        Singleton singleton = Singleton.Instance;
    }
}
```

So, right on this line of code, the *getter* is run (for the first time), checks that the instance is null (which it is at this very moment) and then creates the instance and returns it. From this point forward it should be clear that any other reference to **Singleton.Instance** will just return the instance.

POSSIBLE CONFUSION WHEN NOT USING “NEW” KEYWORD

It’s important to remember that since there is no “**new**” keyword being used wherever the instance is being created, it could look just like an ordinary assignment of a previously existing object. So you just have to remember that the getter is doing all of the work creating the object instance.

Let’s write this example slightly differently to hammer the point home:

```
//Singleton Example
class Singleton
{
    public int Counter;

    ... (previous code left untouched) ...
}

class Program
{
    static void Main()
    {
        Console.WriteLine(Singleton.Instance.Counter++); //0
        Console.WriteLine(Singleton.Instance.Counter++); //1
    }
}
```

This example looks even less intuitively like an instance is being created. However, of course *it is being created* in the first **WriteLine()** because the Instance property is being referenced (creating the instance) and then immediately moves to setting the value for that instance’s Counter.

Of course it should be clear that the instance is not null by the time the second **WriteLine()** is called and from this point forward we’re just purely referencing the instance.

EXPLICIT METHODS IN LIEU OF PROPERTY (NOT RECOMMENDED)

Because code such as this could make it confusing exactly when and where an instance is being created rather than just referenced, you might see some people use explicit methods instead of just a getter:

```
class Singleton
{
    private Singleton() { }

    private static Singleton _instance;

    public static void CreateInstance()
    {
        if (_instance == null)
            _instance = new Singleton();
    }

    public Singleton GetInstance()
    {
        if (_instance != null)
            return _instance;
        else
            return null;
    }
}
```

This makes creating vs returning the instance much more explicit but really adds other problems into the mix. For example: you can run **CreateInstance()** many times over and in different places and not necessarily know that it's doing absolutely nothing at all.

The advice here would be to stick with a *getter* in a Property (or, if you prefer, use a single method for creating/returning the instance exactly as you would use a getter) as we saw in the first example. And perhaps rely on naming by, for example, explicitly using the word “Instance” for the Property’s name.

```
//From now on, pure references to the instance will be referred to as “singleton”
Singleton singleton = Singleton.Instance;
```

It’s also important to note at this point that there’s no way to reach members like “Counter” other than through the static Singleton instance (unless the Counter is static as well, which might cause problems).

SIDE BENEFIT OF A STATIC SINGLETON PROPERTY

Besides the typical case of limiting object creation, the other common benefit to using the Singleton pattern is encapsulating that single object in a static property. Being static, it then becomes *globally accessible*. This makes the object easier to pass around and reference throughout your code.

An example of this usage might be that you’re creating a single-player game and therefore only need a single instance of the class that manages your player1 character. You can create that class using a

Singleton and easily make reference to its static property throughout your code. Which is particularly handy considering that it's likely many classes will want to know about your player1 character.

WHY NOT JUST MAKE THE CLASS STATIC?

The obvious question might be why not just make a static class instead of a Singleton? After all, this would solve the problem of global accessibility pretty easily and appears, on paper, to be exactly why a static class exists in the first place.

The simple answer is that a static class *can't be treated as an instance*. You can't pass it around like an object, but rather can only reference its members. You also, therefore, can't make an interface that can be used to make various different implementations of that class.

This could pose a problem if your single player game has a Singleton for the player class, but you want to have different implementations that each represent different possible control schemes, player states, etc. That wouldn't be possible with a static class unless every possible configuration is housed in a giant monolithic player class. That would be a bad way to go about things if you want to add different player implementations at a later date.

REAL WORLD EXAMPLE: MOCK GAME ENGINE

Let's build a practical mock example of the Singleton pattern using the Game Engine analogy.

For this mock example, we'll suppose that we're working within a Game Engine that can have a single **player1** character that itself can hold multiple **forms** (a Wizard, Knight, etc).

Since we know this ahead of time, It will be convenient to provide a globally accessible object to other classes that need access to this player. And because we're using a Singleton and not simply a static class to provide this, we can also make use Polymorphism and keep *different implementations* for each form of the Player1 character (which holds its own particular controls, attack characteristics and other properties).

Before we define the Singleton classes for each player1 form, let's abstract away common properties and methods so we can polymorphically change between each player1 form:

```
public abstract class AbstractPlayer
{
    public abstract string Name { get; set; }

    public virtual void Attack()
    {
        Console.WriteLine(Name + " - Attacked!");
    }
}
```

Now we'll define the Singletons for each form. Note that in a real example these classes would contain their own unique implementation details. Here's a simplification:

```

//Singleton for PlayerForm1 "Knight"
public class PlayerForm1 : AbstractPlayer
{
    private PlayerForm1() { }

    public override string Name { get; set; } = "Player - Knight";
    private static PlayerForm1 _instance;
    public static PlayerForm1 Instance
    {
        get
        {
            if (_instance == null)
                _instance = new PlayerForm1();
            return _instance;
        }
    }
}

//Singleton for PlayerForm2 "Wizard"
public class PlayerForm2 : AbstractPlayer
{
    private PlayerForm2() { }

    public override string Name { get; set; } = "Player - Wizard";
    private static PlayerForm2 _instance;
    public static PlayerForm2 Instance
    {
        get
        {
            if (_instance == null)
                _instance = new PlayerForm2();
            return _instance;
        }
    }
}

```

Now wherever player1 is referenced we know that we're dealing with only a single object that can be globally referenced. And because it's a Singleton rather than a static class, we can benefit from all sorts of Polymorphic behavior:

```

class Program
{
    static void Main()
    {
        AbstractPlayer player1;

        player1 = PlayerForm1.Instance;
        player1.Attack();

        player1 = PlayerForm2.Instance;
        player1.Attack();
    }
}

```

ASYNCHRONOUS GOTCHA

Now, that should seem like the end of the story regarding Singletons. However, there's a hidden problem with this basic implementation of the pattern. That problem is that there still lingers *a possibility that more than one instance of the Singleton might be created*.

Basically, when running multi-threaded (Asynchronous) code it's *theoretically possible* that two different threads of execution might access the Instance property at exactly the same time. Therefore, it's possible that *both* threads will consider the Singleton instance null (on first access) and then each thread will create a separate instance.

This is what's referred to as "Not Thread Safe" code and would clearly break the entire point of using a Singleton pattern to begin with.

You can avoid this problem by simply doing away with multithreaded code. But this isn't always a realistic plan. What you will probably want instead is a **thread-safe** implementation of the Singleton pattern.

THREAD SAFETY BY NOT LAZILY LOADING THE SINGLETON

You could solve this problem by loading the instance right as the program starts, for example by explicitly loading the instance in the **main()** method before any multi-threaded execution is run. This "**Eager Loading**" typically creates the object and keeps it in memory throughout the lifetime of the running application.

By contrast, loading the object "lazily" at some unknown point during runtime could run into the Asynchronous gotcha problem. However, **Lazy Loading** is an important attribute to keep in principle.

Let's say, for sake of argument, you have 1000's of *different* Singleton's all loading right as the program starts and continuing in memory for the duration of the program running. Or perhaps, for some reason, you're loading a Singleton that takes up a considerable amount of resources and yet isn't going to be needed throughout the lifetime of your running program. Either case could result in potentially a lot of wasted resources just to avoid the problem of thread safety.

STATIC INITIALIZATION PARTIAL SOLUTION

reference: <https://msdn.microsoft.com/en-us/library/ff650316.aspx>

We can go part of the way to solving this problem by forcing initialization of the Singleton instance *immediately on class load* (essentially whenever any member of the class is referenced).

This is slightly less "lazy loaded" since the instance might be created at an unknown point, however possibly when we don't really need it (if we, say, run a method in the class that has no reference to the instance at all). However, it does provide for better thread-safety and clears the ambiguity surrounding instantiation that we saw earlier. We can achieve this with a few minor modifications to our earlier example:

```

public sealed class Singleton
{
    private Singleton() { }

    private static readonly Singleton _instance = new Singleton();
    public static Singleton Instance
    {
        get
        {
            return _instance;
        }
    }
}

```

Let's break down how this all works. We force instantiation of the Singleton on class reference by using the **"new"** keyword in the backing field. We also add the **readonly** keyword to provide an extra protection against the instance being re-initialized somewhere else in the class (by accident, for example). And we simply return the backing field without requiring a null-check since we know we get an instance immediately on class load (before the getter even runs for the first time).

This example is a sort of middle-ground between lazy loading and ensuring thread safety. In most cases it will work just fine, however to ensure absolute thread safety you will have to use additional **"locking"** mechanisms. Here is the example given from MSDN:

```

public sealed class Singleton
{
    private Singleton() { }

    private static volatile Singleton instance;
    private static object syncRoot = new Object();

    public static Singleton Instance
    {
        get
        {
            if (instance == null)
            {
                lock (syncRoot)
                {
                    if (instance == null)
                        instance = new Singleton();
                }
            }
            return instance;
        }
    }
}

```

The explanation for this code above, again from MSDN:

*“This approach ensures that only one instance is created and only when the instance is needed. Also, the variable is declared to be **volatile** to ensure that assignment to the instance variable completes before the instance variable can be accessed. Lastly, this approach uses a **syncRoot** instance to lock on, rather than locking on the type itself, to avoid deadlocks.*

*This double-check locking approach solves the thread concurrency problems while avoiding an exclusive lock in every call to the **Instance** property method. It also allows you to delay instantiation until the object is first accessed. In practice, an application rarely requires this type of implementation. In most cases, the static initialization approach is sufficient.”*

WRAP-UP SUMMARY

Remember that the main problems that Singletons solve are: (A) global static access point for a commonly accessed object, (B) limiting an object to exactly one instance.

Also keep in mind the gotchas and things to consider: (A) Awareness of Instance creation, (B) Ensuring performance using Lazy Loading (when important) and (C) Be aware of Thread Safety.

QUIZ QUESTIONS

A Singleton is preferable to a simple static class because it:

1. Creates an instance
2. Can support multiple implementations (IE: Interface)
3. Isn't loaded lazily
4. Is more performant

The key parts to a Singleton are:

1. Private constructor, static instance property, null check on the instance
2. Private constructor, lazy loading, getter
3. Thread safe, global access, counter
4. None of the above

Is it OK to have two instances of a Singleton class created?

1. No
2. Yes (in all cases)
3. Yes (in some special cases such as multi-threading)
4. Probably not

A developer should strongly consider using the Singleton pattern when

1. Only a single object instance is needed
2. Performance optimization is paramount
3. An object is being “lazily loaded” at runtime
4. Multiple classes refer to a particular object

Why does a Singleton use a private constructor?

1. To do a null check
2. To prevent instantiation
3. To prevent outside classes from instantiating it
4. To keep the instance privately accessible

Multi-threaded apps pose a problem to simple versions of the Singleton patterns because

1. They can’t only produce single instances of objects
2. Each thread always produces a copy of the Singleton object
3. Separate threads might access the instance property at the same time
4. Separate threads will access the instance property at the same time

A typical Singleton may have an ambiguous

1. Point of instantiation
2. Point of reference
3. Value
4. Object

Which Singleton implementation is the least “lazily loaded”?

1. An instance property assigned with a new object after a null check.
2. An instance backing field directly assigned with a new object.
3. An method returning a new object after a null check
4. They’re all just as equally lazy

QUIZ ANSWERS

1. **(2) Can support multiple implementations (IE: Interface)**
2. **(1) Private constructor, static instance property, null check on the instance**
3. **(1) No**
4. **(4) Multiple classes refer to a particular object**
5. **(3) To prevent outside classes from instantiating it**
6. **(3) Separate threads might access the instance property at the same time**
7. **(1) Point of instantiation**
8. **(2) An instance backing field directly assigned with a new object.**

3. Prototype Pattern

PATTERN TYPE: Creational

FREQUENCY OF USE: Medium

MEMORY HELPER: Clone Objects

PATTERN OBJECTIVE: Use a low-cost method of creating new objects by cloning existing objects.

PREAMBLE: The Prototype pattern is very simple to create and understand. The foundation for this pattern relies on inheritance of a virtual/abstract method that allows for **cloning** of the enclosing type.

SCAFFOLD EXAMPLE

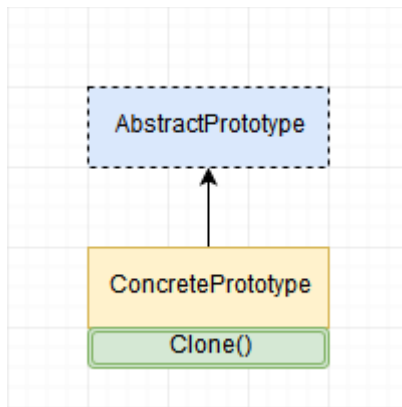
Our scaffold for this pattern starts with an abstract class containing the abstract **Clone()** method. It also contains a default constructor that subclasses can reference through inheritance:

```
abstract class AbstractPrototype
{
    public string Name;
    public AbstractPrototype(string Name)
    {
        this.Name = Name;
    }
    public abstract AbstractPrototype Clone();
}
```

Our first subclass passes the constructor value back to the the base class constructor and implements and override of the **Clone()** method:

```
class ConcretePrototype1 : AbstractPrototype
{
    public ConcretePrototype1(string Name) : base(Name) { }
    public override AbstractPrototype Clone()
    {
        return (ConcretePrototype1)this.MemberwiseClone();
    }
}
```

The following diagram depicts how this pattern is structured:



SHALLOW VS DEEP COPY

The **MemberwiseClone()** method is available to all objects in .NET and simply returns a *shallow copy* of the current class object (referenced by the “this” keyword). A “shallow copy” simply means an *object reference* is returned rather than a *new* object. It’s important to keep this in mind as any changes to reference properties of the clone would reflect back to the original object. We’ll take a closer look at this caveat in a moment.

We’ll use the **main()** method as a consumer class for the code we have so far:

```
static void Main()
{
    AbstractPrototype prototype1 = new ConcretePrototype1("Product A");
    AbstractPrototype clone1 = prototype1.Clone();
    Console.WriteLine(prototype1.Name + " cloned to another " + clone1.Name);
}
```

SHALLOW COPY CAVEAT

To illustrate the potential problem with using shallow copies of objects, let’s consider what happens if we were to change the value of the **Name** string in either the Prototype or Clone object (indicated in red below):

```
static void Main()
{
    AbstractPrototype prototype1 = new ConcretePrototype1("Product A");
    AbstractPrototype clone1 = prototype1.Clone();
    clone1.Name = "Product A - Reburished";
    Console.WriteLine(prototype1.Name + " cloned to another " + clone1.Name);
}
```

You might have expected, both, **Prototype** and **Clone** to output “Product A - Refurbished” given that the clone is simply a reference copy of the original object. However, **strings** are treated like value *types* in the .NET Framework so each object’s string retains its own values. The trouble is that other object types that are treated by *reference* will not behave like this. Consider what would happen if we had an object reference embedded in each prototype/clone object instead:

```
abstract class AbstractPrototype
{
    public SomeReferenceType SomeReferenceType = new SomeReferenceType();
    public string Name;
    public AbstractPrototype(string Name)
    {
        this.Name = Name;
    }
    public abstract AbstractPrototype Clone();
}

class SomeReferenceType { public int x; }
```

When we run the following code, we see that changing the **SomeReferenceType.x** value, in either the Clone or the Prototype, changes the value in the other object:

```
static void Main()
{
    AbstractPrototype prototype1 = new ConcretePrototype1("Product A");
    AbstractPrototype clone1 = prototype1.Clone();
    prototype1.SomeReferenceType.x = 5;
    clone1.SomeReferenceType.x = 9;
    Console.WriteLine(prototype1.SomeReferenceType.x); //shows “9”
}
```

If you need to perform a *deep copy*, which essentially creates an entirely new object with it’s own memory addresses, the required overhead may make the Prototype pattern less appealing. The simplest way to perform a deep copy would be to specifically overwrite the object reference(s) in the cloned object using the “new” keyword:

```
class ConcretePrototype1 : AbstractPrototype
{
    public ConcretePrototype1(string Name) : base(Name) { }
    public override AbstractPrototype Clone()
    {
        AbstractPrototype deepClone = (ConcretePrototype1)this.MemberwiseClone();
        deepClone.SomeReferenceType = new SomeReferenceType();
        return deepClone;
    }
}
```

REAL WORLD EXAMPLE

For a real world scenario we're going to imagine that we're creating a bullet manager in a shoot-em-up game. If the game were to spit out and destroy bullets many times per-second, this could result in poor performance.

So, to solve the problem we'll implement a Prototype **IBullet** that we can simply clone once the manager's bullet cache dips below a certain threshold. We start by defining a simple interface for our bullets and create our concrete bullet types that implement it:

```
interface IBullet
{
    IBullet Clone();
}
class ExplodingBullet : IBullet
{
    public IBullet Clone()
    {
        return (ExplodingBullet)this.MemberwiseClone();
    }
}
class LaserBullet : IBullet
{
    public IBullet Clone()
    {
        return (LaserBullet)this.MemberwiseClone();
    }
}
```

Next, we'll define a **BulletManager** class that stores a cache of bullets. And since we've made the BulletManager generic we can create a cache of bullets of whichever type we want as long as it's an IBullet.

```
class BulletManager<T> where T : IBullet, new()
{
    public IBullet[] Cache;
    public BulletManager(int initialStock)
    {
        Cache = new IBullet[initialStock];
        for (int i = 0; i < initialStock; i++)
            Cache[i] = new T();
    }
}
```

Then we'll have a Fire() method that removes bullets from the cache as they're being fired. If the cache dips below 7 bullets in total, it will clone the last bullet in the cache and add it to a new slot in the cache:

```

class BulletManager<T> where T : IBullet, new()
{
    ...
    public void Fire(int amount)
    {
        for (int i=0; i < amount; i++)
        {
            Array.Resize(ref Cache, Cache.Length - 1);
            Console.WriteLine(Cache[0].GetType().Name +
                " Fired. Current Stock: " + Cache.Length
            );

            if (Cache.Length < 7)
            {
                Array.Resize(ref Cache, Cache.Length + 1);
                Cache[Cache.Length - 1] = Cache[Cache.Length - 2].Clone();
                Console.WriteLine(Cache[Cache.Length - 1].GetType().Name +
                    " Added. Current Stock: " + Cache.Length
                );
            }
        }
    }
}

```

In the **main()** method we create the BulletManager instance, supplying it with 9 bullets, and then fire 5 of them. This will result in 3 bullets being cloned and added back into the cache.

```

static void Main()
{
    BulletManager<LaserBullet> emitter = new BulletManager<LaserBullet>(9);
    emitter.Fire(5);
}

```

QUIZ QUESTIONS

Which scenario sounds like the best candidate for the Prototype pattern?

1. An application that allows the user to copy/paste data
2. A game that is still in prototype and very rough
3. An application that can process thousands of objects of the same basic type
4. An application that has many products with small differences

If a shallow copy of an object has an integer property that is changed, what happens to the prototype version?

1. It stays the same
2. It's integer property changes to the same value
3. It's integer property might change or it might not, it depends.
4. It will be destroyed

Would it be OK to use a Prototype pattern in tandem with a Factory or Singleton pattern?

1. Yes
2. No
3. It's probably OK, but will likely be too complicated

What of the following would be a prime reason for using the Prototype pattern?

1. Simplifying object creation
2. Your application has a lot of objects
3. Your classes are very simple in structure
4. Increasing your app's performance

Which of the following sounds most accurate? All objects treated as prototypes:

1. Require copies that end up exactly the same
2. Should be thought of as templates for later copies
3. Should later become better defined
4. Will ultimately be discarded when a copy is made

What is an interesting aspect of Strings that have an implication on cloned versions

1. Strings are really just an array of individual characters
2. Strings are value types that are treated like reference types
3. Strings are reference types that are treated like value types
4. Strings that change in value change other strings

If your application has a lot of reference types embedded in cloned copies

1. It will be difficult to perform deep copying
2. You will have to find a way to turn them into value types
3. Creating copies could be more costly than it's worth
4. You probably shouldn't use the Prototype pattern

QUIZ ANSWERS

1. **(3) An application that can process thousands of objects of the same basic type**
2. **(1) It stays the same**
3. **(1) Yes**
4. **(4) Increasing your app's performance**
5. **(2) Should be thought of as templates for later copies**
6. **(3) Strings are reference types that are treated like value types**
7. **(3) Creating copies could be more costly than it's worth**

4. Factory Method Pattern

PATTERN TYPE: Creational

FREQUENCY OF USE: High

MEMORY HELPER: Hiding Object Creation

PATTERN OBJECTIVE: Use an interface method for creation of simple objects that share that interface, rather than explicitly using the “new” keyword to create objects wherever clients need them. This keeps object creation details from the rest of the system, facilitating the open/closed principle.

PREAMBLE: The Factory Method pattern is a creational pattern geared towards instantiation of objects in a system. It's particularly well suited for scenarios where the system won't always know the specific type of objects ahead of time, and yet needs to seamlessly adapt to new objects being added into it later on.

The system in question could be as simple as one that's used by one small company having “**Product**” objects that need to be created (including “products” that aren't yet offered).

For example, a company that sells *books* today, and say *eBooks* tomorrow, will want the part of the software that depends on the **BookProduct** object to not have to change wherever these objects are created. At the same time, they'll want the software to easily accommodate new **eBookProduct** objects.

By using the Factory Method pattern to underpin this object creation, it becomes a simple case of having an **eBookProduct** class inherit from a common “**Product**” interface and created in a related “**Factory**” class.

Among other benefits, this satisfies the **open/closed** principle in software development. This principle roughly entails having a system that can extend to new classes that share a common interface, yet closed for changes to the broader system dependent on those unforeseen classes.

While demand for the open/closed principle is important even in simple examples such as with the hypothetical Book Company, it would be even *more crucial* in something like a **Framework** that is open to user-defined object creation procedures that cannot be fully anticipated. The rest of the system needs to know how to cope with these unforeseen objects, and yet stay rigidly closed to any fundamental changes in how it handles these objects.

Factory Method Scaffolding

The first thing we need to get a handle on is how a basic Factory Method pattern is constructed, focusing on the scaffolding and ignoring specific details for the moment. The pattern can be broken up into three main groups of classes/interfaces:

- **Product Classes/Interface**
- **Factory Classes/Interface**
- **Client Class**

Let's start by defining a simple "product" class:

```
class ConcreteProductA { }
```

We're naming the product class in a very generic way and appending "A" because we know that we'll probably want to add more product classes later, appending "B", "C" and so forth. It may not be the best naming in a production situation, but it will hopefully communicate the basic concept.

Then, considering that the product class will have a somewhat common instantiation procedure throughout the system - and needing to be created often or in many places - we'll define a factory that handles the task of pumping out instances of the product.

```
class ConcreteFactoryA
{
    public ConcreteProductA CreateProduct()
    {
        return new ConcreteProductA();
    }
}
```

Now that we have a product and a factory, we need a client. For simplicity we'll just designate the main() method as the client:

```
class Program
{
    static void Main()
    {
        ConcreteProductA productA = new ConcreteFactoryA().CreateProduct();
    }
}
```

Remember that one of the main points of having a Factory Method is to hide details of product creation from clients throughout the system. We've gotten close to satisfying this task, but the client still knows too much about the exact - in other words, "concrete" - type of object being created. To really give this code utility we have to hide behind interfaces/abstraction. So let's make a few changes:

```

/* Products */
abstract class AbstractProduct
{
    protected AbstractProduct()
    {
        Console.WriteLine(GetType() + " Created!");
    }
}

class ConcreteProductA : AbstractProduct { }

/* Factories */
interface IFactory
{
    AbstractProduct CreateProduct();
}

class ConcreteFactoryA : IFactory
{
    public AbstractProduct CreateProduct()
    {
        return new ConcreteProductA();
    }
}

```

Now, the client doesn't have to know anything directly about the product when it comes to product creation (for now). The client only needs to know the common creation method `CreateProduct()`, and not any of the instantiation details contained within it:

```

/* Client */
class Program
{
    static void Main()
    {
        IFactory FactoryA = new ConcreteFactoryA();
        AbstractProduct ProductA = FactoryA.CreateProduct();
    }
}

```

Also, any new product which follows a similar procedure can easily be added into the system (in red):

```

class Program
{
    static void Main()
    {
        IFactory FactoryA = new ConcreteFactoryA();
        IFactory FactoryB = new ConcreteFactoryB();
        AbstractProduct ProductA = FactoryA.CreateProduct();
        AbstractProduct ProductB = FactoryB.CreateProduct();
    }
}

/* Products */
abstract class AbstractProduct
{
    protected AbstractProduct()
    {
        Console.WriteLine(GetType() + " Created!");
    }
}

class ConcreteProductA : AbstractProduct { }
class ConcreteProductB : AbstractProduct { }

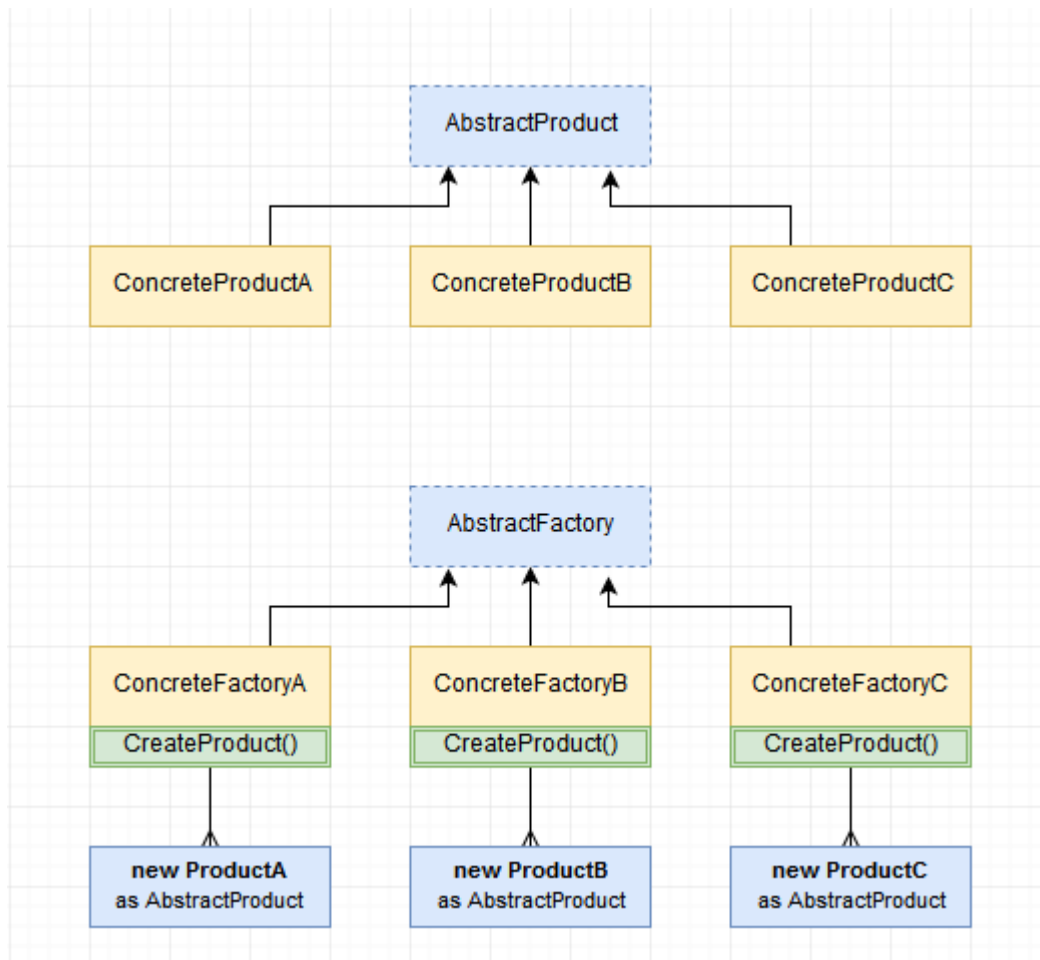
/* Factories */
interface IFactory
{
    AbstractProduct CreateProduct();
}

class ConcreteFactoryA : IFactory
{
    public AbstractProduct CreateProduct()
    {
        return new ConcreteProductA();
    }
}

class ConcreteFactoryB : IFactory
{
    public AbstractProduct CreateProduct()
    {
        return new ConcreteProductB();
    }
}

```

Here's a handy visualization representing the important relationship between Products and Factories:



To get a better understanding of how this might produce a system-wide benefit, let's convert this scaffolding into something resembling a real scenario. Let's say you're writing an application that draws objects - for whatever purpose - and you know ahead of time that every object will have its own specific drawing algorithm. However, you're not sure what objects might later be added into the system so you want to make sure it's open for these additions. We start by defining shapes which all will share a common Draw() interface:

```
interface Shape
{
    void Draw();
}
```

```

class Circle : Shape
{
    public void Draw()
    {
        Console.WriteLine("Drawing a Circle...one moment please...");
    }
}

class Oval : Shape
{
    public void Draw()
    {
        Console.WriteLine("Drawing an Oval...one moment please...");
    }
}

class Square : Shape
{
    public void Draw()
    {
        Console.WriteLine("Drawing a Square...one moment please...");
    }
}

class Rectangle : Shape
{
    public void Draw()
    {
        Console.WriteLine("Drawing a Rectangle...one moment please...");
    }
}

```

Now, you decide to define creator factories depending on - for whatever reason - whether or not the shapes contain angles or rounded edges. Here we let the factory instantiate the shape based on a string value passed into the CreateShape() method:

```

interface FactoryCreator
{
    Shape CreateShape(string shapeType);
}

```



```

class FactoryRounded : FactoryCreator
{
    public Shape CreateShape(string shapeType)
    {
        if (shapeType == "circle")
            return new Circle();
        else if (shapeType == "oval")
            return new Oval();
        else
            throw new Exception();
    }
}

class FactoryCornered : FactoryCreator
{
    public Shape CreateShape(string shapeType)
    {
        if (shapeType == "square")
            return new Square();
        else if (shapeType == "rectangle")
            return new Rectangle();
        else
            throw new Exception();
    }
}

```

And in the client class, we bring everything together:

```

class Program
{
    static void Main(string[] args)
    {
        Shape square1 = new FactoryCornered().CreateShape("square");
        square1.Draw();
        Shape oval1 = new FactoryRounded().CreateShape("oval");
        oval1.Draw();
    }
}

```

Extending With More Shapes

If we decide to add more - or more complex shapes - later on into the system, we simply add more factory and shape classes deriving from the existing **FactoryCreator** and **Shape** interfaces. We can then sleep easy knowing the system - the clients that rely on Shape objects - will be able to handle new objects (EG: a "**FactoryPointed**" class that creates pointy **Triangle** object) in the same way as before.

Now, if this code was written for a *Framework* - a system that has to be open for extensibility but closed for change - then maybe the example we looked at here isn't the best way of handling a drawing system such as this.

For example, you might want to redefine what "Cornered" or "Rounded" shapes are, or are able to include, but as it stands you would have to create entirely new factories to handle expanded definitions.

So, if you want to add spirals as an object created in **FactoryRounded** (because it is technically a shape that has no corners or points but is indeed round) you wouldn't - or, rather, shouldn't - be able to add it to the **FactoryRounded** class directly, but instead create a new one entirely, perhaps called **FactorySpiraled** that implements the same type of interface.

```
class FactorySpiraled : FactoryCreator
{
    public Shape CreateShape(string shapeType)
    {
        if (shapeType == "spiral")
            return new Spiral();
        else
            throw new Exception();
    }
}
```

Or perhaps better yet, we could make the **CreateShape()** methods *virtual* thereby allowing the user to extend the existing Factory classes as well as the creation methods. The ability to override the main creation methods for a factory in order to create another, specialized version, is one of the key advantages of using the Factory Method pattern extendable systems and frameworks. Once again, the key point here is uniformity and flexibility:

```
class FactoryRounded : FactoryCreator
{
    public virtual Shape CreateShape(string shapeType)
    {
        ... //original details unchanged
    }
}

class FactoryRoundedUserDefined : FactoryRounded
{
    public override Shape CreateShape(string shapeType)
    {
        if (shapeType == "spiral")
            return new Spiral();

        return base.CreateShape(shapeType);
    }
}
```

WRAP-UP SUMMARY

Factory Method is a great way to deal with delegating object creation to Factory subclasses, keeping the rest of the system untouched should changes arise. It adapts well to objects that share the same interface. If your application doesn't have a common creation procedure that you can hide behind an interface, then it may not be the best pattern for the task. For this reason it isn't common to see the Factory Method in places where you need parameterized constructors handling many different ways that the objects could be constructed.

Considering, once again, the previous example: if we were to create objects with very different properties, this might not be the best way to do it. Sticking with the Shape-Drawing Application analogy, if you want to pass in details of object properties at object creation (**radius**, **length**, **height**, etc) then it becomes clear that rounded objects and cornered objects won't have the same interface (cornered objects don't have a use for radius).

QUIZ QUESTIONS

Factory Method is named after

1. A method of creating Factory classes
2. Abstract Factories
3. Factories that use a common method for object creation
4. Factories that use many different methods for creating objects

Factory Method hides the creation of "Product" objects behind

1. An abstract class
2. An Interface
3. A common Factory abstract class/interface method
4. Other classes

A big part of Factory Method is to reduce the amount of dependencies in a system:

1. True
2. False

The open/closed principle basically states:

1. Developers should be open minded, not closed minded
2. Keeping a broader system closed for extensibility, while open to change
3. Keeping a broader system closed for change, while open for extensibility
4. A system should never be closed, and always open to changes

Using a common method for object creation in the system means

1. If object creation details change, you'll have to change the method calls
2. If object creation details change, it will be hidden behind the method details
3. You potentially lose the power and versatility of Constructors
4. Both B and C are true

A Factory Method focuses on

1. Object creation only
2. System Structure only
3. Mostly Object creation, but also has Structural benefit
4. Mostly Structure, but also has Object creation benefits

If a new product is added into the system using the Factory Method pattern, the most basic question to ask is

1. Should you put it in its own Factory class or in an existing Factory class?
2. Does it fit the interface of other similar objects in the system?
3. Do you really need the product to begin with?
4. Should you hide the product behind an Abstract class or an Interface?

Factory Method is very common with

1. Drawing objects
2. Making products
3. Closed interfaces
4. Extendable Frameworks

Because the Factories use methods to create objects this opens the door to

1. Allowing objects to be created wherever you want without issue
2. Extending to new, specialized factories that override the base creation methods
3. Never using constructors ever again
4. Eliminating dependencies when using the "new" keyword

QUIZ ANSWERS

1. **(3) Factories that use a common method for object creation**
2. **(3) A common Factory abstract class/interface method**
3. **(1) True**
4. **(3) Keeping a broader system closed for change, while open for extensibility**
5. **(4) Both B and C are true**
6. **(3) Mostly Object creation, but also has Structural benefit**
7. **(2) Does it fit the interface of other similar objects in the system?**
8. **(4) Extendable Frameworks**
9. **(2) Extending to new, specialized factories that override the base creation methods**

5. Abstract Factory Pattern

PATTERN TYPE: Creational

FREQUENCY OF USE: High

MEMORY HELPER: Abstracting Creation of Related Objects

PATTERN OBJECTIVE: Use Abstract/Interface definitions or creation of simple objects that have different interfaces and yet need to “interact” with one another, keeping the entire creation/interaction process highly abstracted.

PREAMBLE: In this lesson we'll look at the "**Abstract Factory**" pattern. It bears some resemblance to the **Factory Method** pattern but has a somewhat different purpose and execution that distinguishes it. A good place to start is by looking at the similarities it shares with Factory Method.

Both patterns rely on classes of related "**Products**" being pumped out by "**Factory**" classes that hide the exact nature of the objects behind interfaces for, both, the products and the factories. This achieves a high degree of abstraction that cascades throughout the system, limiting dependencies throughout it.

The third key element in both patterns is the "**Client**" class which calls the factory methods that create the required product objects. The main difference between Abstract Factory and Factory Method is that the Abstract Factory Client class manages the product objects being created all without knowing their exact underlying types. In other words, the Client is blindly managing the objects as completely abstract entities, simply calling their methods and properties exposed by their abstract definitions and interfaces.

Also different is that the factories create sets of product objects that have different interfaces and yet interact with each other in some way. The client, meanwhile, keeps reference to these products by holding onto them through Aggregation (roughly defined as class-level objects who's "lifetime" is managed elsewhere - in this case, the factories that produced them).

SETTING UP THE SCAFFOLDING

Let's start by creating the familiar "Product" scaffolding. In this case, we'll start with two separate product interfaces. This allows products from the "**A**" set to interact with products from the "**B**" set, and vice versa:

```

//PRODUCT INTERFACES
interface IProductA
{
    void Interact(IProductB productB);
}

interface IProductB
{
    void Interact(IProductA productA);
}

```

IProductA's are a set of objects that can interact with a **IProductB** set of objects. If you need a mental picture, just imagine any set of related objects in the real world that commonly interact with another set of related objects. For example, **IProductA** could represent a set of *acidic chemicals* while **IProductB** could be a set of *base chemicals*.

Now we need to construct the definitions for the actual products that fit these interfaces. These definitions are intentionally simplistic. In a real scenario they would otherwise hold meaningful differences (and yet still be able to be hidden behind common interfaces):

```

//PRODUCTS FROM "A" SET
class ProductA1 : IProductA
{
    public void Interact(IProductB productB)
    {
        Console.WriteLine("Interact " + this.GetType().Name +
            " with " + productB.GetType().Name);
    }
}
class ProductA2 : IProductA
{
    public void Interact(IProductB productB)
    {
        Console.WriteLine("Interact " + this.GetType().Name +
            " with " + productB.GetType().Name);
    }
}
//PRODUCTS FROM "B" SET
class ProductB1 : IProductB
{
    public void Interact(IProductA productA)
    {
        Console.WriteLine("Interact " + this.GetType().Name +
            " with " + productA.GetType().Name);
    }
}

```

```

class ProductB2 : IProductB
{
    public void Interact(IProductA productA)
    {
        Console.WriteLine("Interact " + this.GetType().Name +
            " with " + productA.GetType().Name);
    }
}

```

Now that we have our product scaffolding, let's move on to constructing the factory scaffolding. We begin by defining the factory interface that all factories share.

```

//FACTORY INTERFACE
interface IFactory
{
    IProductA CreateProductA();
    IProductB CreateProductB();
}

```

In this case, each factory will simply output one possible combination of products from the “**IProductA**” set and “**IProductB**” set via the methods “**CreateProductA()**” and “**CreateProductB()**”. Here are the concrete factories that do exactly that:

```

//FACTORIES
class ConcreteFactoryA : IFactory
{
    public IProductA CreateProductA()
    {
        return new ProductA1();
    }
    public IProductB CreateProductB()
    {
        return new ProductB1();
    }
}

class ConcreteFactoryB : IFactory
{
    public IProductA CreateProductA()
    {
        return new ProductA2();
    }
    public IProductB CreateProductB()
    {
        return new ProductB2();
    }
}

```


Now that we have our products and factories fully defined, we'll want to bring them together in a client class. Pay special attention to the fact that the client knows nothing about the exact products it will receive other than their basic Interfaces:

```
class Client
{
    private IProductA productA;
    private IProductB productB;

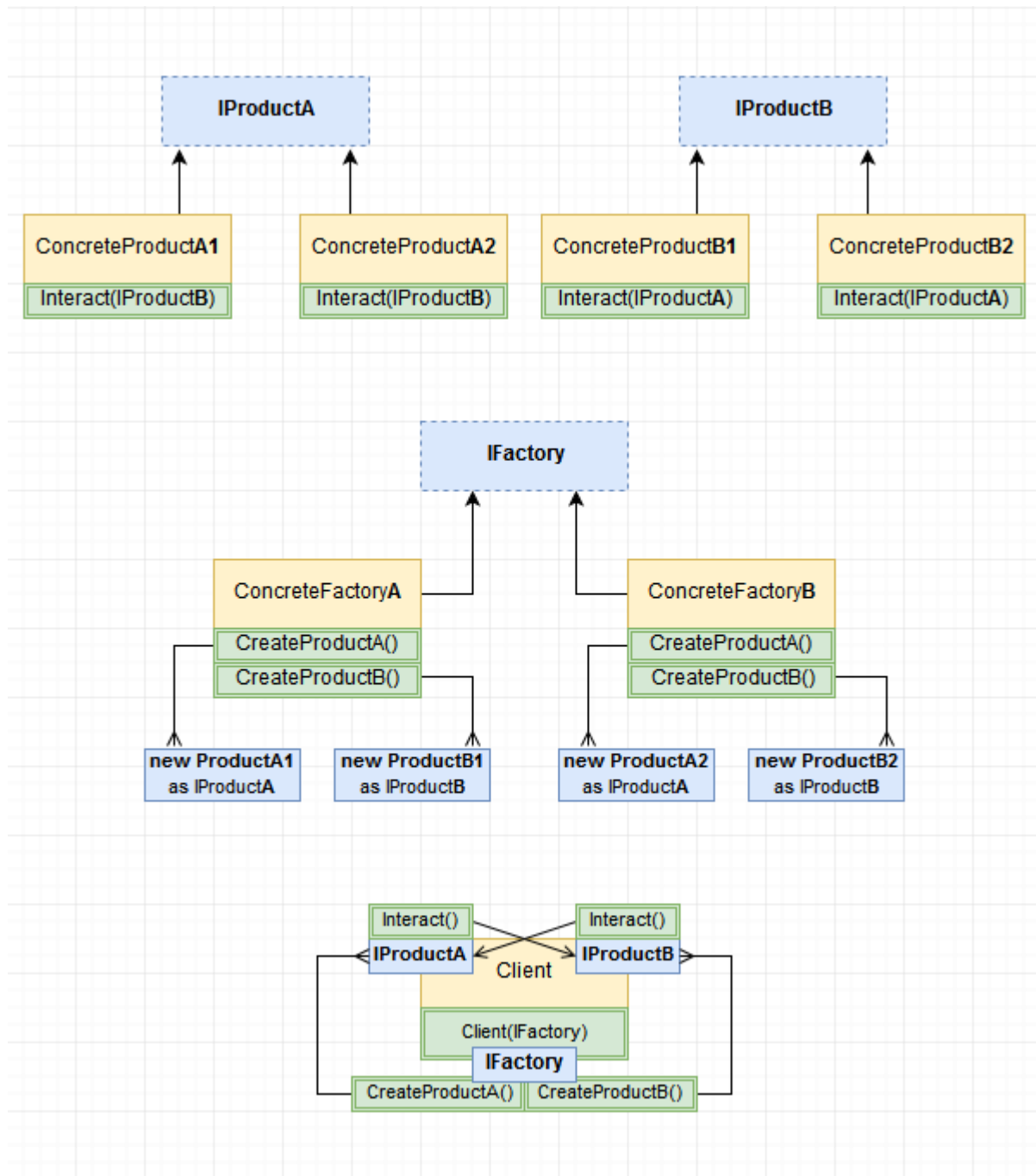
    public Client(IFactory factory)
    {
        productA = factory.CreateProductA();
        productB = factory.CreateProductB();
    }

    public void DoInteraction()
    {
        productB.Interact(productA);
    }
}
```

The client here takes in an **IFactory** via its constructor and assigns its private fields of type **IProductA** and **IProductB** based on whatever products are output by the exact Factory passed in.

The client can also manage the interaction between the products with the public **DoInteraction()** method. Again, the key point here is the client is able to do its business without knowing the full details of what it's receiving in terms of factories and products.

Below is a visual representation of the entire structure and relevant associations. Take special note to the particularly high level of abstraction at all levels when creating object dependencies, but particularly in the Client:



UNDERSTANDING THE CLIENT'S TASK BY ANALOGY

This is a crucial part of what makes the design pattern “Abstract.” By abstracting the information away, we de-couple the classes. This is so important in software development that we, the developers, are willing to create this complicated overhead in order to keep the system clean. Consider an analogy for why this is useful.

Imagine that Client is something like a Chemist's assistant. The Chemist has an important job but he isn't monolithic - he can't do everything - so he hands off key tasks to various assistants (client classes).

The Chemist wants the tasks done by the assistants while giving them as little information as possible. The assistants, in other words, shouldn't have to know exactly how what they're doing works or get confused by extra information, else they might get their own ideas and create a disaster.

Instead, they should rely on the simple labels and instructions given to them and mix the chemicals accordingly before handing them off someplace else.

With that analogy in mind, the final piece of the puzzle is letting the client do its basic task:

```
class Program
{
    public static void Main()
    {
        Client client1 = new Client(new ConcreteFactoryA());
        client1.DoInteraction();
    }
}
```

REAL WORLD EXAMPLE

One of the difficulties of working with this design pattern is that it can be *very* abstract. To help make it more concrete we'll apply the scaffolding code to a hypothetical business application. For this example project we're writing software for a company that sells and rents musical instruments. The system needs to handle one set of "products," - guitars and keyboards - as well as another set - rental order forms and sale order forms.

Each combination of guitar/keyboard, sale/rental is pumped out by a dedicated factory, while the client class is able to process the relationship between the instrument/transaction without knowing the details.

As before, we'll start by defining the abstract classes for the set of "A" products (instruments):

```
abstract class AbstractInstrument
{
    protected string Manufacturer;
    public abstract void SetManufacturer(string manufacturer);
    public abstract string GetManufacturer();
}
```

And also the abstract class for the set of “B” products (forms):

```
abstract class AbstractOrderForm
{
    public abstract void Assign(AbstractInstrument instrument, string manufacturer);
}
```

Then we'll create our concrete instrument product classes:

```
class Guitar : AbstractInstrument
{
    public override void SetManufacturer(string manufacturer)
    {
        Manufacturer = manufacturer + " Guitar";
    }
    public override string GetManufacturer()
    {
        return Manufacturer;
    }
}
```

```
class Keyboard : AbstractInstrument
{
    public override void SetManufacturer(string manufacturer)
    {
        Manufacturer = manufacturer + " Keyboard";
    }
    public override string GetManufacturer()
    {
        return Manufacturer;
    }
}
```

And then we define our concrete order form product classes:

```
class SaleForm : AbstractOrderForm
{
    public override void Assign(AbstractInstrument instrument, string manufacturer)
    {
        instrument.SetManufacturer(manufacturer);
        Console.WriteLine(
            String.Format("{0} created for a sold {1} of type {2}",
this.GetType().Name, instrument.GetType().Name, instrument.GetManufacturer())
        );
    }
}
```

```

class RentalForm : AbstractOrderForm
{
    public override void Assign(AbstractInstrument instrument, string manufacturer)
    {
        instrument.SetManufacturer(manufacturer);
        Console.WriteLine(
            String.Format("{0} created for a rented {1} of type {2}",
this.GetType().Name, instrument.GetType().Name, instrument.GetManufacturer())
        );
    }
}

```

Now, for our Factories, we define a simple interface for creating a product of each type in each factory:

```

interface IFactory
{
    AbstractInstrument CreateInstrument();
    AbstractOrderForm CreateForm();
}

```

We'll have four possible combinations of instrument/order form, producing either a guitar sale, guitar rental, keyboard sale or keyboard rental. Our concrete factories manage the creation of each possible combination:

```

class GuitarSale : IFactory
{
    public AbstractInstrument CreateInstrument()
    {
        return new Guitar();
    }
    public AbstractOrderForm CreateForm()
    {
        return new SaleForm();
    }
}

class GuitarRental : IFactory
{
    public AbstractInstrument CreateInstrument()
    {
        return new Guitar();
    }
    public AbstractOrderForm CreateForm()
    {
        return new RentalForm();
    }
}

```

```

class KeyboardSale : IFactory
{
    public AbstractInstrument CreateInstrument()
    {
        return new Keyboard();
    }
    public AbstractOrderForm CreateForm()
    {
        return new SaleForm();
    }
}

```

```

class KeyboardRental : IFactory
{
    public AbstractInstrument CreateInstrument()
    {
        return new Keyboard();
    }
    public AbstractOrderForm CreateForm()
    {
        return new RentalForm();
    }
}

```

Our client class then manages every order by taking in a factory, holding reference to the products it creates and “interacting” them by calling the Assign() method to complete the order:

```

class Order
{
    private AbstractInstrument instrument;
    private AbstractOrderForm orderForm;
    public Order(IFactory factory)
    {
        instrument = factory.CreateInstrument();
        orderForm = factory.CreateForm();
    }
    public void CompleteOrder(string manufacturer)
    {
        orderForm.Assign(instrument, manufacturer);
    }
}

```

We can then use the client to complete the order:

```
class Program
{
    public static void Main()
    {
        Order order1 = new Order(new GuitarSale());
        order1.CompleteOrder("Ibanez");
        Order order2 = new Order(new KeyboardRental());
        order2.CompleteOrder("Roland");
    }
}
```

WRAP-UP

in this example we used only 4 possible combinations of products. If our product set would grow much larger and the amount of possible interactions increased, our code could certainly get considerably more complicated. This is one of the downsides of using the pattern. A real world scenario would likely use a more robust inheritance scheme than you see here but since production scenarios vary wildly, it is most important to understand the basics of the pattern in as simple of an example as possible.

QUIZ QUESTIONS

Abstract Factories rely on

1. Overriding base methods to create new types of objects
2. A client that abstracts creation of related objects
3. Interfaces only
4. Abstract classes only

An Abstract Factory and Factory Method share this in common

1. Use the same interface for all products
2. They both specialize in the creation of objects
3. Client class, Product class, Factory class
4. Both B and C are true
5. Both A and C are true

Factory Method and Abstract Factory are completely different and cannot be combined:

1. True
2. False

The client in an Abstract Factory holds reference to products via:

1. Inheritance
2. Polymorphism
3. Aggregation
4. Encapsulation

The type of factories we made create products that have

1. The same object creation procedure
2. The ability to interact
3. The exact same type
4. The same interface

The following best describes the role of the Client:

1. Holds reference to products without knowing their exact type
2. Needs to know everything about factories and products
3. Only cares about interacting one product with another
4. Only cares about creating products

QUIZ ANSWERS

1. **(2) A client that abstracts creation of related objects**
2. **(4) Both B and C are true**
3. **(2) False**
4. **(3) Aggregation**
5. **(2) The ability to interact**
6. **(1) Holds reference to products without knowing their exact type**

6. Structural Wrappers – Preface pt.1

INTRO TO STRUCTURAL 'WRAPPER' PATTERNS

Moving on from **Creational** patterns - which are focused on *creation of objects* - we'll now focus on **Structural** patterns - which focus on structuring code with different *implementation details*.

The first set of Structural patterns we'll look at share a lot in common, *so much so that they can be hard to distinguish from one another*. Generally speaking, the commonality comes from focusing on abstracting classes as simple "**Wrappers**" around other classes - typically wrapping around an external library.

Wrappers are meant to supply a layer that hides the classes and methods they're wrapping around - the sub-layer. If it's done right, the rest of the system doesn't know the intimate details of the sub-layer, allowing it to be swapped out or have method internals re-written without fear of breaking the rest of the system that depends on the sub-layer.

The design patterns we'll be looking at in the next few lessons simply differ in their main purpose for wrapping. To make the point clear as to the value of using wrappers we'll examine a common scenario and then see how different wrapper-like patterns approach solving the issue.

For this scenario, imagine you are writing software and are not sure of which logging framework you want to end up using. You have two main candidates, **LogFrameworkTypeA** and **LogFrameworkTypeB**, that each have very different implementation details. And yet you want to be able to go ahead and construct the system around it without directly caring about which framework you end up using.

We'll pretend that the main logging methods for either Framework can be represented as simply as follows:

```
class LogFrameworkTypeA
{
    public void Log(string message)
    {
        Console.WriteLine("Logging Framework #1 Outputs: " + message);
    }
}

class LogFrameworkTypeB
{
    public void Log(bool error)
    {
        Console.WriteLine("Logging Framework #2 Outputs Errors = " + error);
    }
}
```

Each Framework has a different method signature for their main logging method and yet we'll want to be able to represent them using a common **Interface** to make them interchangeable. Using an **object** as

input parameter - a base type for both strings and bools - will do the trick of representing logging behavior of either type of logging Framework:

```
interface ILog
{
    void Log(object message);
}
```

Next, we'll want to define our wrapper classes that will hide the details of the underlying Framework being used.

```
class LogWrapperA : ILog
{
    private LogFrameworkTypeA externalModule;
    public LogWrapperA()
    {
        externalModule = new LogFrameworkTypeA();
    }
    public void Log(object message)
    {
        string msg = message as string;
        externalModule.Log(msg);
    }
}

class LogWrapperB : ILog
{
    private LogFrameworkTypeB externalModule;
    public LogWrapperB()
    {
        externalModule = new LogFrameworkTypeB();
    }
    public void Log(object message)
    {
        bool error = ((string)message == "Fail") ? true : false;
        externalModule.Log(error);
    }
}
```

You'll notice the **Log()** implementation details are very different in either case, however the rest of the system doesn't need to know that, and will operate the same way regardless of which is used:

```
public static void Main()
{
    ILog loggerA = new LogWrapperA();
    loggerA.Log("Success");

    ILog loggerB = new LogWrapperB();
    loggerB.Log("Success");
}
```

To swap out your preferred logging Framework, you can now freely change all `new LogWrapperA()` references to `new LogWrapperB()` and vice versa.

WRAPPER PREFACE WRAP-UP

In the next set of lessons we'll adapt this basic scenario of a fictional logging system, fit into wrapper-like patterns that handle details differently from one another. These wrapper-like patterns are:

- **Proxy**
- **Decorator**
- **Facade**
- **Adapter**
- **Bridge**

7. Structural Wrappers – Preface pt.2

DIFFERENTIATING WRAPPER PATTERNS

Before we look more closely at the five main wrapper patterns, let's briefly define how they differ from one another.

Proxy – A typical proxy wrapper is a class that looks, to a consuming client, like whatever class it's wrapping around. Both the real class as well as the proxy class implement the same interface. The proxy class is usually a very simple wrapper with perhaps only adding protection mechanisms.

Facade – A Façade normally is used to hide complexity of a set of modules. This could take the form of hiding multiple complex objects within a wrapper class that calls multiple methods behind single, simplified methods. For example, if there is a bunch of methods/modules involved when displaying output the screen, it could be better to just include these under a single Display() method that the rest of the system calls.

Adapter – An Adapter wraps a class that has an interface, or concrete definition, that doesn't match exactly with what the rest of the system expects. The Adapter contains the Adaptee and 'converts' the non-fitting interface to one that does fit the rest of the system. More concrete than a Bridge, primarily used for wrapping 3rd party modules.

Bridge – A Bridge is a bit like a more complex Adapter. It attempts to abstract, both, the Wrapper and the underlying class being wrapped. Since abstraction is a key feature of this pattern, it usually does not work when adapting 3rd party modules as you would not be able to define the interfaces for the 3rd party classes.

With these definitions in mind, let's look once again at the wrapper we wrote in the previous lesson and see where it would best fit as representing one of these wrapper patterns.

First, we have two external class (3rd part) modules that basically do the same thing but in different ways:

```
class LogFrameworkTypeA
{
    public void Log(string message)
    {
        Console.WriteLine("Logging Framework #1 Outputs: " + message);
    }
}

class LogFrameworkTypeB
{
    public void Log(bool error)
    {
        Console.WriteLine("Logging Framework #2 Outputs Errors = " + error);
    }
}
```

We then formed wrapper classes around each module and used a common interface to represent either module to the rest of the system:

```
interface ILog
{
    void Log(object message);
}

class LogWrapperA : ILog
{
    private LogFrameworkTypeA externalModule;
    public LogWrapperA()
    {
        ...
    }
    public void Log(object message)
    {
        ...
    }
}

class LogWrapperB : ILog
{
    private LogFrameworkTypeB externalModule;
    public LogWrapperB()
    {
        ...
    }
    public void Log(object message)
    {
        ...
    }
}
```

WHICH PATTERN IS IT?

It isn't a simple **Proxy**, because there's a bit more complexity than what would define that pattern.

It can't be a **Decorator**, because there isn't extra functionality being extended in the wrapping class.

It can't be a **Façade** as we're not including multiple modules under a single wrapper, or else simplifying a complex set of features behind a single wrapping class.

It seems to fit the definition of an **Adapter** quite well. It satisfies the need to "fit" otherwise non-fitting interfaces with the rest of the system, in a sense "adapting" outside code to the system around it. However, could it be a **Bridge** considering that there is abstraction involved, most notably by using a common interface `ILog`?

While it is important to note that real-world code doesn't always fit squarely with the definition of any single pattern, the best way to answer this is to look at the *intent* of each pattern.

If a bridge has the main intent of abstracting all relevant classes from the ground-up, it wouldn't fit our scenario very well as it depends on 3rd party code that we can't completely abstract away. In other words, each wrapping class has a concrete dependency to either `LogFrameworkTypeA` or `LogFrameworkTypeB` because each of those are external modules. We can't directly abstract them behind a common interface. So, that means our example is probably more like the Adapter pattern, which is designed to fit classes/modules (usually third party modules) that don't share a common interface.

The main intent of an Adapter pattern is to fit things outside our control that otherwise don't fit, whereas a bridge is more of an abstract way of building a system. Where an adapter is forced to have more concrete dependencies, a Bridge pattern has no such constraint and is more abstract. Usually the defining difference is whether or not you have control over defining abstraction of the classes being wrapped around. In other words, it depends on whether you're *adapting* outside code or *bridging* the gap between similar classes that have different functionality in your own code. In this sense, you could look at our example as a slightly more abstracted Adapter.

In the next lesson we will look at more closely at the most basic version of Adapter pattern.

8. Adapter Pattern

PATTERN TYPE: Structural

FREQUENCY OF USE: High

MEMORY HELPER: Module Converter

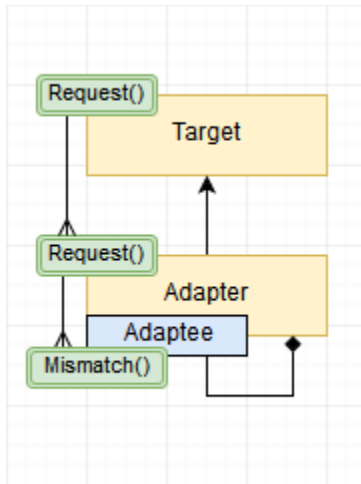
PATTERN OBJECTIVE: Concretely retrofit a module within a system that expects a different interface.

PREAMBLE: An adapter pattern is comprised of a simple interplay between three classes:

- Target
- Adapter
- Adaptee

```
class Target
{
    public virtual void Request()
    {
    }
}
class Adapter : Target
{
    private Adaptee _adaptee = new Adaptee();
    public override void Request()
    {
        _adaptee.MismatchingRequest();
    }
}
class Adaptee
{
    public void MismatchingRequest()
    {
    }
}
class Program
{
    static void Main()
    {
        Target target = new Adapter();
        target.Request();
    }
}
```


The **Target** class usually would represent an existing class that has been set up to work in a particular way, and yet doesn't match how the new **Adaptee** class operates (which is usually an inalterable external module). The **Adapter** comes into the middle and bridges that mismatch:



ADAPTERS: A VISUAL ANALOGY

We see *actual* Adapters everywhere in the real world. Adapters (converters, etc) are a way of connecting a “Target” with an “Adaptee” that don’t immediately fit together. Think of a phone charge cable with a USB “Adaptee” that doesn’t fit a wall socket “Target.” In this case you would need a usb-to-wallsocket *Adapter*. Let’s convert this analogy into a problem that the Adapter pattern can solve.

Suppose we have a Target class that a system has been built around and expects a **Log()** method that takes in a message of type **string**, but our chosen logging platform Adaptee offers a **Logger()** method that takes in type **bool**. Here is how an Adapter can solve this mismatch problem: create an Adapter as a subclass of the Target class and include an instance of the Adaptee in the Adapter, wrapping its logging method in one that fits the rest of the system.

ADAPTER EXAMPLE IN ACTION

Recalling that an Adapter is usually constrained by the definitions of 3rd party modules, here is a more concrete example of an Adapter than what we created in our preface.

First, we have the 3rd party module Adaptee we can’t change and yet need to fit into our existing Target class:

```
class LogFrameworkAdaptee
{
    public void Logger(bool error)
    {
        Console.WriteLine("Logging Framework Outputs Errors = " + error);
    }
}
```

Then we have the Target class itself, which we'll heavily simplify here as:

```
class LogTarget
{
    public virtual void Log(string message)
    {
        Console.WriteLine("LogTarget Log() Called: " + message);
    }
}
```

We can effectively fit everything together by inheriting from the Target and include an instance of the Adaptee in the Adapter class:

```
class LogAdapter : LogTarget
{
    private LogFrameworkAdaptee externalModule = new LogFrameworkAdaptee();
    public override void Log(string message)
    {
        base.Log(message);
        bool error = (message == "Fail") ? true : false;
        externalModule.Logger(error);
    }
}
```

Existing references to the LogTarget now need to simply reference the LogAdapter instead

```
class Program
{
    public static void Main()
    {
        LogTarget logger = new LogAdapter();
        logger.Log("Success");
    }
}
```

ADAPTER WRAP UP

As you can see, the main difference with our previous example is that everything is concretely defined here. That's because usually we are using the Adapter pattern to fit two classes that cannot be changed or extend from any kind of abstraction. This means that we are, usually, only creating the Adapter class while the Adaptee and Target had already existed and cannot be easily changed.

For this reason this pattern is usually constructed as very *concrete* and not relying on any kind of abstraction. Because our original scenario had some flexibility (recalling that we were in control of creating the "Target" as ILog) for the purpose of swapping underlying Adaptee's, it isn't exactly like the usual Adapter pattern.

QUIZ QUESTIONS

An Adapter class typically wraps around

1. A mismatching interface
2. An instance of the Target
3. An instance of the Adaptee
4. A Framework

An Adapter uses the following to “merge” the Target and Adaptee

1. Inheritance and Composition
2. Interfaces and Abstraction
3. Log() and Logger() methods
4. None of the above

Which of these best describes the intent of the Adapter pattern?

1. Platform independence
2. Futureproofing code
3. Retrofitting new code
4. None of the above

An Adapter usually can't directly rely on inheritance because

1. The Adaptee class is likely not in our control
2. The Adaptee class has a different interface
3. The Adaptee class is a module
4. An adapter has to be concrete

Which of the following is an analogous real-world example of the Adapter pattern

1. A power extension cord
2. A saddle on a horse
3. Snow tires on a car
4. A travel power converter

QUIZ ANSWERS

1. **(3) An instance of the Adaptee**
2. **(1) Inheritance and Composition**
3. **(3) Retrofitting new code**
4. **(1) The Adaptee class is likely not in our control**
5. **(4) A travel power converter**

9. Bridge Pattern

PATTERN TYPE: Structural

FREQUENCY OF USE: Medium

MEMORY HELPER: Platform Independent Abstraction

PATTERN OBJECTIVE: To abstract classes we are creating in order to separate the abstraction from the implementation, allowing for portability/swapability.

BRIDGE vs ADAPTER

A Bridge pattern is very similar in implementation to the Adapter pattern except whereas an Adapter is more *concrete*, a Bridge is more *abstract*. The typical case in which you'll find the Bridge pattern is in portable code, or code that has different implementation details depending on the underlying platform. In essence, a bridge is written into the codebase from the ground-up to accommodate many different possible usage scenarios whereas an Adapter typically retrofits new code into an existing codebase.

For this reason, you might see a Bridge most often in Frameworks and systems that need lateral extensibility; that is to say, situations where you need to create independent *implementation hierarchies* of the main abstraction.

BRIDGE SCAFFOLDING

Our scaffolding starts with an arbitrary number of possible concrete implementations that fit a common interface:

```
interface AbstractImplementation
{
    void Operation();
}

class ImplementationA : AbstractImplementation
{
    public void Operation()
    {
        Console.WriteLine("ImplementationA Operation");
    }
}
```

```

class ImplementationB : AbstractImplementation
{
    public void Operation()
    {
        Console.WriteLine("ImplementationB Operation");
    }
}

```

We then define the base abstraction for the wrapper class that can swap either implementation through its member reference:

```

abstract class BaseAbstraction
{
    public AbstractImplementation Implementation;

    public virtual void Operation()
    {
        Implementation.Operation();
    }
}

class RefinedAbstraction : BaseAbstraction
{
    public override void Operation()
    {
        Implementation.Operation();
    }
}

```

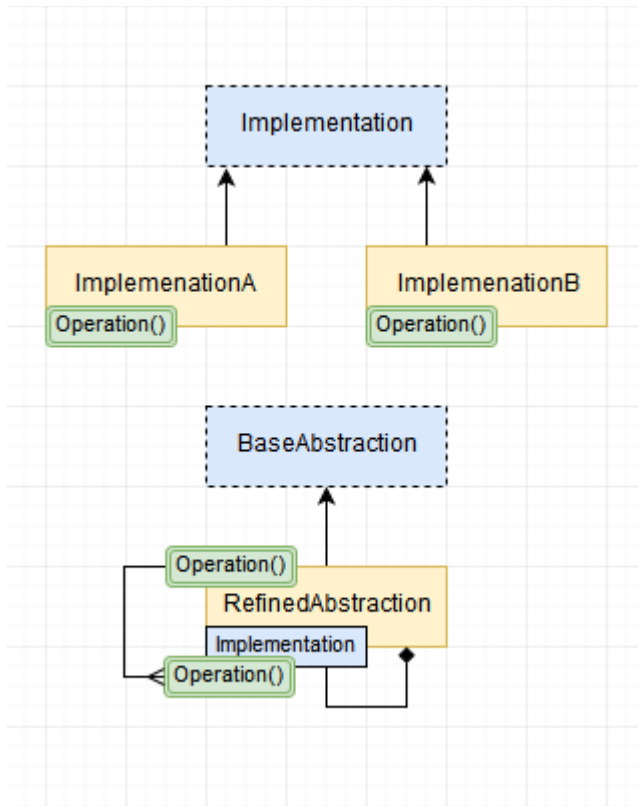
The particular implementation dependency can now be arbitrarily determined in a single consuming class:

```

class Program
{
    static void Main()
    {
        BaseAbstraction Flexible = new RefinedAbstraction();
        Flexible.Implementation = new ImplementationA();
        Flexible.Operation();
        Flexible.Implementation = new ImplementationB();
        Flexible.Operation();
    }
}

```

Here's a visualization of the heart of this pattern, keeping in mind that the RefinedAbstraction has an Implementation of any derived subclass Implementation:



LOGGING FRAMEWORK USING A BRIDGE

Recalling the previous “Logging Framework” examples we had looked at, how would a Bridge pattern fit in with that scenario? If we pivot from imagining ourselves as *consumers* of the Logging Framework - trying to retrofit it into our code - and instead look at it from the perspective of *creators* of the Framework itself, we can imagine using a Bridge to create independent implementations depending on the consumer platform.

For example, as creators of this hypothetical Logging Framework we know that our end-users want to use objects and methods that basically do the same thing. However, we also know that each end-user platform might have it's own underlying peculiarities in dealing with that functionality.

We can therefore use a Bridge pattern to create any number of implementations that share a common interface and can be determined at runtime.

```

interface ILogImplementation
{
    void Log(string message);
}

class LogImplementationA : ILogImplementation
{
    public void Log(string message)
    {
        Console.WriteLine("Logging Framework for Windows Outputs: " + message);
        //Do Windows Specific Logging Here
    }
}

class LogImplementationB : ILogImplementation
{
    public void Log(string message)
    {
        Console.WriteLine("Logging Framework for Linux Outputs: " + message);
        //Do Linux Specific Logging Here
    }
}

```

This first point is the single most crucial difference between this scenario and the earlier ones we saw as *consumers* of the Framework. Previously, we hadn't been able to directly define this interface as these different implementations would have belonged to 3rd party code outside of our control. Now that we have the interface that abstracts particular logging implementations we can whip up a simple wrapper to form an extra level of abstraction:

```

//BaseAbstraction
interface ILogAbstraction
{
    ILogImplementation Module { get; set; }

    void Log(string message);
}

//RefinedAbstraction
class LogWrapper : ILogAbstraction
{
    public ILogImplementation Module { get; set; }

    public void Log(string message)
    {
        Module.Log(message);
    }
}

```


The Wrapper inherits from the main Interface abstraction and is often referred to as the “Refined Abstraction” for this pattern. This high degree of abstraction at the implementation and wrapper levels allows us to decouple dependencies and allow for effortless swapping of implementations in consuming classes:

```
class Program
{
    public static void Main()
    {
        string platform = "windows";
        ILogAbstraction logger = new LogWrapper();
        switch (platform)
        {
            case "windows":
                logger.Module = new LogImplementationA();
                break;
            case "linux":
                logger.Module = new LogImplementationB();
                break;
        }
        logger.Log("Success");
    }
}
```

In the above code we “hard-coded” the destination platform, however, in an actual real-world scenario we could gather that information and determine the destination platform at runtime.

WRAP UP

We have seen how the Adapter and Bridge patterns solve similar problems by wrapping around code that doesn’t “fit” the existing code base (Adapter) or needs flexible implementations (Bridge). We have also seen scenarios that don’t fit either patterns perfectly which is an important reminder that patterns are just good starting points for solving common problems, not concrete solutions that fit every situation. It’s more important to identify the core problem that your software needs to solve, rather than worry about which pattern fits the task exactly. In the next lesson, we’ll look at another similar looking pattern: Proxy.

QUIZ QUESTIONS

Which of the following is a likely candidate for the Bridge pattern?

1. An application that needs to swap out 3rd party modules
2. An application that needs to have extensible implementations
3. Both A and B
4. None of the above

What's the cornerstone of the Bridge pattern?

1. Interfaces and Abstraction
2. A method with the same name
3. Inheritance trees
4. None of the above

Which of these best describes the intent of the Bridge pattern?

1. Platform independence
2. Retrofitting new code
3. Abstraction at all levels
4. None of the above

A Bridge usually requires the following

1. Two or more separate implementations
2. 3rd party modules
3. Full control over the classes it involves
4. Interfaces

A Bridge should make an exact implementation

1. Supported by all platforms
2. Not depend on platforms
3. Unknown to the client
4. Variable at runtime

QUIZ ANSWERS

1. **(2) An application that needs to have extensible implementations**
2. **(1) Interfaces and Abstraction**
3. **(1) Platform independence**
4. **(3) Full control over the classes it involves**
5. **(3) Unknown to the client**

10. Proxy Pattern

PATTERN TYPE: Structural

FREQUENCY OF USE: Medium

MEMORY HELPER: Gatekeeper wrapper

PATTERN OBJECTIVE: Provide a surrogate or placeholder for another object to control access to it.

HOW IS A PROXY DIFFERENT?

After having looked at the Bridge vs Adapter patterns the Proxy pattern might look extremely simple by comparison. Unlike those other patterns, a Proxy isn't about abstraction, retrofitting or anything complex. The main intent of a Proxy is basically just a simple Wrapper that acts purely as a gatekeeper to the class it is wrapping around. There are a variety of scenarios for why you would want to have a simple gatekeeper proxy controlling access to another class, usually including some added functionality in the wrapping class. Here are some such scenarios, a few of which we'll later demonstrate in our code examples:

Virtual Proxy: Defer the wrapped classes object initialization, IE: Lazy Loading

Remote Proxy: A local object that represents a remote resource, acting as a stand-in object until the remote object becomes available.

Protection Proxy: Provide access control to an underlying sensitive object, such as operating system objects.

PROXY SCAFFOLDING

We start with the familiar interface that is common to the "real subject" as well as the Proxy that wraps around the real subject's object:

```
interface Subject
{
    void Request();
}
```

We then define the real subject class that implements this interface

```
class RealSubject : Subject
{
    public void Request()
    {
        Console.WriteLine("RealSubect's Real Request() called");
        //Request's implementation details
    }
}
```

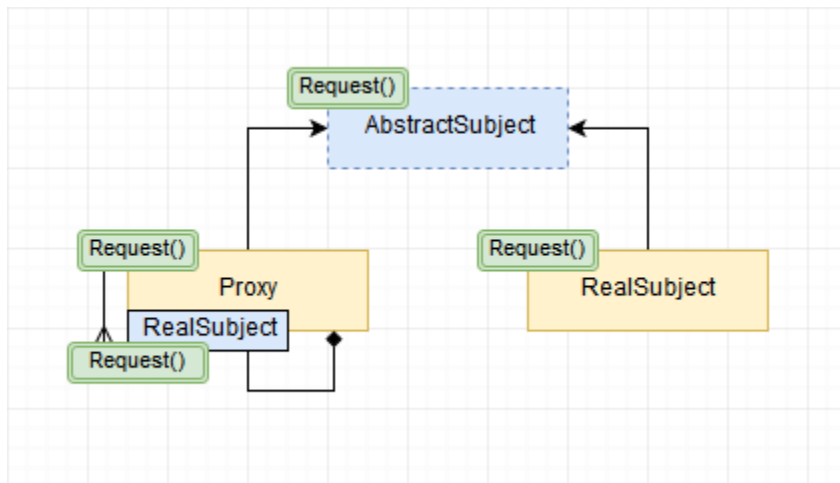
And, once again very familiar to us by now, we construct the Proxy wrapper class that contains the real subject and fits the same interface:

```
class Proxy : Subject
{
    private RealSubject realSubject;
    public Proxy()
    {
        realSubject = new RealSubject();
    }
    public void Request()
    {
        realSubject.Request();
    }
}
```

Predictably, we can now use the Proxy to control access to the underlying real subject:

```
class Program
{
    static void Main()
    {
        Subject proxy = new Proxy();
        proxy.Request(); //calls RealSubject's Request
    }
}
```

The following is a visual of the basic class relationships and keeping in mind that the Client would instantiate the Proxy to get at the RealSubject:



At this point, the Proxy seems close to the simplest wrapper class possible, and it essentially is. The important thing now is to determine exactly why the Proxy is controlling access to the underlying object. In other words, what are the implementations details of the Proxy in question? Recalling our previous list of possible added functionality, we'll next look at something resembling a real-world implementation.

REAL WORLD EXAMPLE

In this example, we'll start off with defining the "real subject" class as one that returns an image resource as a URL string while also abiding by an interface.

```
interface Subject
{
    void Request();
    string URL { get; set; }
}

class ImageResource : Subject
{
    public string URL { get; set; }
    public void Request()
    {
        var random = new Random();
        if (random.Next(2) < 1)
            URL = null;
        else
            URL = "Some Image URL";
    }
}
```

An actual **Request()** method would do some sort of data-pulling magic but instead we've simulated this functionality by returning a string if the image URL is properly located, or else returning null if it isn't. We simply randomized the result so that we can test this out later.

Next we'll create our Proxy that wraps around and controls access to the underlying subject class. The first bit of added functionality is we can lazily instantiate the real subject when the **Request()** is actually called. This means that we can instantiate a, presumably, less resource-heavy Proxy and not take up larger resources until it actually fulfills its **Request()** function.

We can achieve this with a simple null check before the actual request is made:

```
class ImageProxy : Subject
{
    public string URL { get; set; }
    private ImageResource imageRes;
    public void Request()
    {
        if (imageRes == null)
            imageRes = new ImageResource();
    }
}
```

The next bit of added functionality is to return a "dummy" image URL if the real Request() returns null

```
class ImageProxy : Subject
{
    ...
    ...
    public void Request()
    {
        if (imageRes == null)
            imageRes = new ImageResource();
        imageRes.Request();
        if (imageRes.URL == null)
            URL = "Dummy Image URL";
        else
            URL = imageRes.URL;
    }
}
```

Meanwhile, the client that makes use of the Proxy doesn't know that any of this extra functionality is being hidden behind the Proxy:

```
class Program
{
    static void Main()
    {
        Subject proxy = new ImageProxy();
        proxy.Request();
        Console.WriteLine(proxy.URL);
    }
}
```

QUIZ QUESTIONS

Which of the following best describes the role of the Proxy:

1. Sanitizes data from the real subject
2. A fake version of class with a similar interface
3. Ensures an object is lazily loaded
4. An intermediary between a client and real subject

Which of the following can be said about a Proxy:

1. Forwards information
2. Provides additional logic
3. Both A and B are true
4. None of the above

What is the main function of a Virtual Proxy?

1. Delays instantiation of the real subject
2. Protects against data breaches
3. Completely hide the real subject
4. None of the above

What is the main function of a Remote Proxy?

1. Retrieve information from the internet
2. Manage a remote resource with a local object
3. Forward information to a remote resource
4. Pass information to a real subject

Which of the Proxy functions did we see in our real world code?

1. Virtual Proxy, Protection Proxy
2. Remote Proxy, Protection Proxy
3. Virtual Proxy, Remote Proxy
4. Virtual Proxy, Protection Proxy and Remote Proxy

QUIZ ANSWERS

1. **(4) An intermediary between a client and real subject**
2. **(3) Both A and B are true**
3. **(1) Delays instantiation of the real subject**
4. **(2) Manage a remote resource with a local object**
5. **(3) Virtual Proxy, Remote Proxy**

11. Facade Pattern

PATTERN TYPE: Structural

FREQUENCY OF USE: High

MEMORY HELPER: Simplification via Composition

PATTERN OBJECTIVE: To provide a simplified class that reduces complexity of subsystem class modules that are brought together in Facade methods to complete more complex tasks. This helps decouple multiple dependencies by holding them in a single class while providing a simple way of “summarizing” a set of more complex tasks.

WHY CHOOSE A FACADE?

A Facade is yet another “wrapper” type of structural design pattern. One of its key differences from other wrappers that we’ve seen is it doesn’t rely on Interfaces or Abstract classes to “hide” information but rather uses simple object composition. Basically the role of the Facade is to encapsulate more complex functionality by containing objects responsible for that functionality and then calling “summary” methods that in turn call (usually more complicated) methods of those containing objects. A Facade might be a good choice whenever you have a routine that depends on a series of objects working in concert towards producing a result you can summarize under a single method call.

For example. Later on in the lesson we’ll look at using a Facade to represent a simplified product ordering scenario. In reality, ordering a product contains at least three main bits of functionality: Checking inventory, accepting payment, and notifying the customer if both are successful.

A Facade can take the three objects responsible for all of this and “hide” them behind a simple “FacadeOrder” class that simply contains a single “PlaceOrder()” method for determining whether an order is successful. This can help the system maintain separation of concern, encourage decoupling by limiting dependencies to a single Facade, and improve code readability.

FACADE SCAFFOLDING

For our scaffolding example we’ll start off with an arbitrary set of three “modules” - which are just a group of classes that don’t necessarily fit a typical interface but will end up working together in some way as a “subsystem”:

```

//SubSystem Modules A, B and C
class ModuleA
{
    public void OperationX()
    {
        Console.WriteLine(" ModuleA Method");
    }
}
class ModuleB
{
    public void OperationY()
    {
        Console.WriteLine(" ModuleB Method");
    }
}
class ModuleC
{
    public void OperationZ()
    {
        Console.WriteLine(" ModuleC Method");
    }
}

```

Then we define the Facade that contains object references to these subsystem modules:

```

//Facade Simplification
class Facade
{
    private ModuleA moduleA;
    private ModuleB moduleB;
    private ModuleC moduleC;

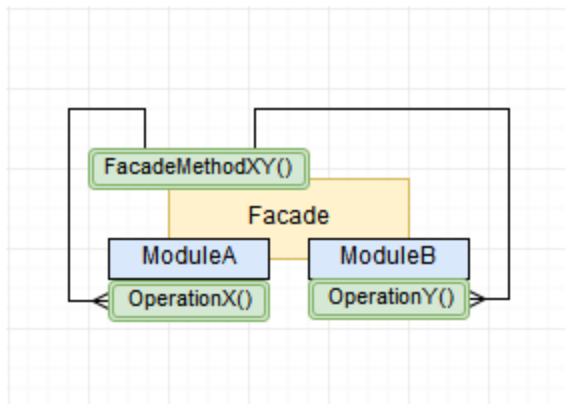
    public Facade()
    {
        moduleA = new ModuleA();
        moduleB = new ModuleB();
        moduleC = new ModuleC();
    }
}

```

Modules that are expected to work in concert to produce a single result can now be hidden behind simplified methods in the Facade:

```
//Facade Simplification
class Facade
{
    ...
    ...
    public void FacadeMethodA()
    {
        Console.WriteLine("FacadeMethodA: ");
        moduleA.OperationX();
        moduleB.OperationY();
    }
    public void FacadeMethodB()
    {
        Console.WriteLine("FacadeMethodB: ");
        moduleC.OperationZ();
    }
}
```

You can visualize the core relationship of encapsulating multiple external module methods in a single Facade method with the following diagram:



REAL WORLD EXAMPLE

The usefulness of this kind of pattern becomes a lot clearer when looking at something closer to a real-world scenario. As mentioned previously, this example will assume we have three subsystem modules for completing the task of ordering a product:

- Inventory
- Payment
- Notify

Only after all three of the conditions for those modules are satisfied can we then say an order is successfully placed. Here is how we can represent those subsystem modules and their main tasks.

We start with defining the Inventory subsystem. It's main task is to simply check the given product's inventory and return that value as an int (randomized for testing purposes):

```
//SubSystems
class Inventory
{
    public int Check()
    {
        Random currentInventory = new Random();
        return currentInventory.Next(4);
    }
}
```

Next, the Payment subsystem's main task is to process payment and return whether or not it was successful:

```
class Payment
{
    public bool Process()
    {
        Random isSuccessful = new Random();
        if (isSuccessful.Next(2) < 1)
            return false;
        else
            return true;
    }
}
```

And finally, the Notification subsystem simply concerns itself with sending email to notify the customer:

```
class Notify
{
    public void SendEmail()
    {
        Console.WriteLine("Email Sent!");
    }
}
```

Now we'll have to define an order processing Facade where all of the subsystem modules come together and determine if we can place an order via a single, simplified method call:

```
//Facade
class FacadeOrder
{
    private Inventory inventory = new Inventory();
    private Payment payment = new Payment();
    private Notify notify = new Notify();

    public void PlaceOrder()
    {
        if (inventory.Check() > 0 && payment.Process())
        {
            notify.SendEmail();
            Console.WriteLine("Order successful! Details have been sent by email.");
        }
        else
            Console.WriteLine("Sorry, your order could not be processed!");
    }
}
```

Placing an order in the rest of the system becomes as simple as:

```
class Program
{
    static void Main()
    {
        new FacadeOrder().PlaceOrder();
    }
}
```

QUIZ QUESTIONS

A Facade's main focus is

1. Composition
2. Abstraction
3. Simplification
4. None of the above

For a Facade to work, modules must

1. Fit a common interface
2. Be numerous
3. Come from external frameworks
4. None of the above

What is a similarity between a Facade and Proxy?

1. Both forward information to the “real” object
2. Both provide a network interface
3. Both don’t contain real functionality other than hiding information
4. None of the above

What can be said of Facade and dependencies

1. It abstracts them from the broader system
2. It can localize and reduce them
3. It contains none of them
4. It’s not a good way of decoupling them

What’s a good real-world analogy for a Facade pattern?

1. An ignition for starting a car
2. A Halloween mask
3. Laminate on furniture
4. A synopsis for a story

You should be able to replace subsystem object of a Facade

1. With any kind of object
2. With an object that shares the same interface
3. Without changing anything else in the Facade
4. Without changing objects that depend on the Facade

QUIZ ANSWERS

1. **(3) Simplification**
2. **(4) None of the above**
3. **(1) Both forward information to the “real” object**
4. **(2) It can localize and reduce them**
5. **(1) An ignition for starting a car**
6. **(4) Without changing objects that depend on the Facade**

12. Decorator Pattern

PATTERN TYPE: Structural

FREQUENCY OF USE: Medium

MEMORY HELPER: Add object functionality dynamically

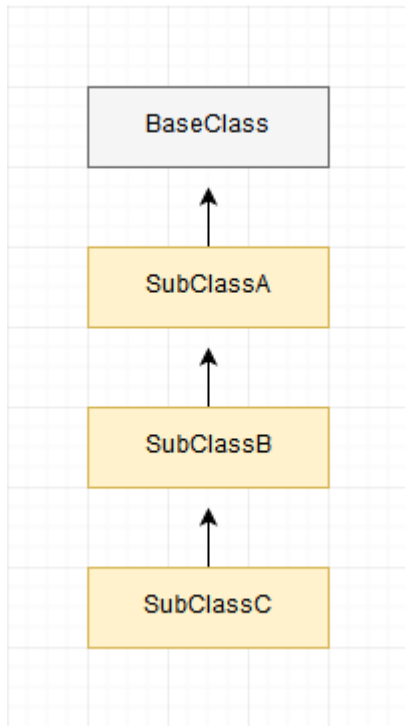
PATTERN OBJECTIVE: To add responsibilities to an object at runtime using composition that follows common inheritance as “components”.

DECORATOR

The Decorator pattern is an interesting type of pattern which combines simplified inheritance with extendable composition to add to class behavior. The essence of a Decorator is to build and add functionality on-the-fly as “decorations” to a single type of object. But before we examine this pattern, we should look at the typical way of approaching building up object complexity using inheritance.

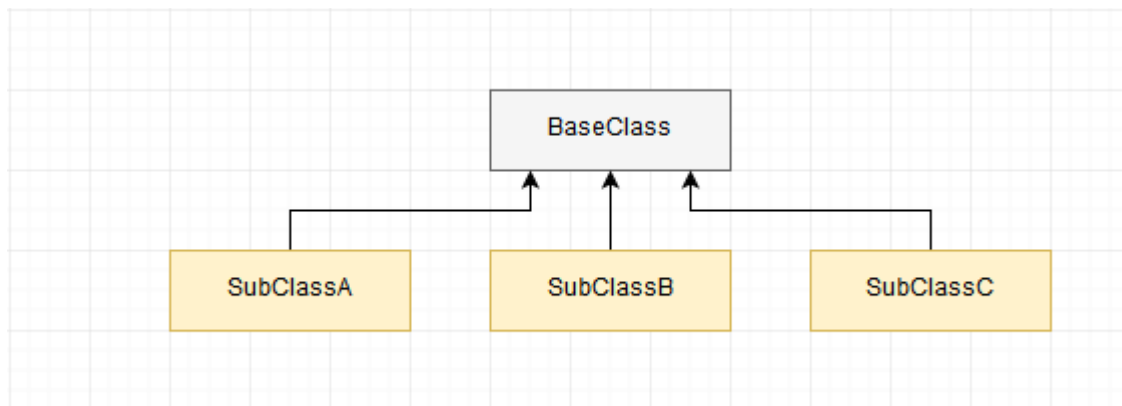
VERTICAL VS HORIZONTAL INHERITANCE TREES

At the most basic level, you can have one of two inheritance schemes. One that is more **vertical** or one that is more **horizontal**. Before choosing which basic scheme to follow, you have to examine the *intent* behind the kind of objects you want to create. The **intent of vertical inheritance** is usually a way of extending and building up a *single kind of object, but with added complexity with every inheriting class* - building a single type of object layer-by-layer using inheritance to add ever-increasing complexity.]This would look something like the diagram below, scaling indefinitely in a vertical direction (arrows indicating inheritance. EG: **SubClassC** inheriting from **SubClassB**, etc).



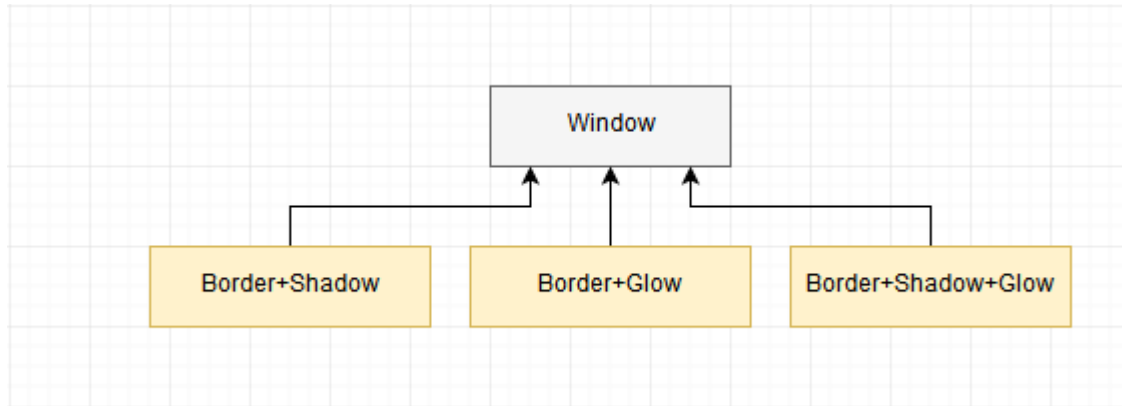
As an example, picture an application Window running on an operating system that uses vertical inheritance to add borders, resizing functions, styles and so forth. We're really picturing a single object (a simple Window) that can be vertically extended to add additional features. We'll revisit this scenario in our example project.

By contrast, **the intent of horizontal inheritance** is geared towards building diverse sets of objects that nevertheless share some common functionality. In this inheritance scheme you would picture a bunch of fundamentally different objects that all inherit from a common base that they all share. It would look something like this:

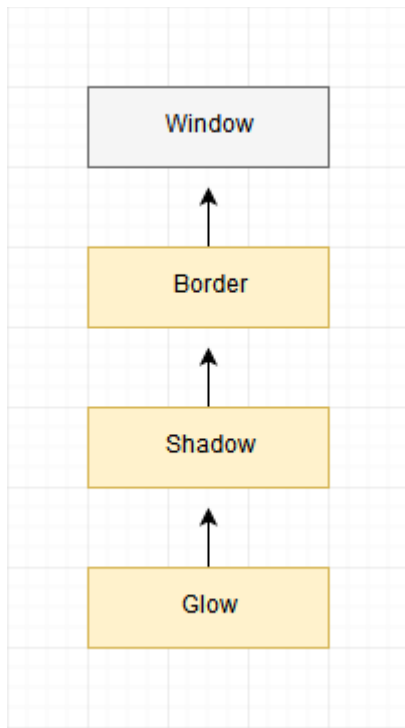


This kind of scheme is well suited for something like a video game where you have a common enemy type that all enemies share from (with shared aspects like attack methods, health, etc) and yet are expected to have very different functionality. You can imagine horizontally building up Knights and Wizards and Goblins, and so forth, that all inherit from a common Enemy class.

But this kind of horizontal inheritance wouldn't suit the Window scenario where we're gradually building up a single object's functionality. Just picture the amount of repeated code that would exist in this horizontal inheritance scenario:



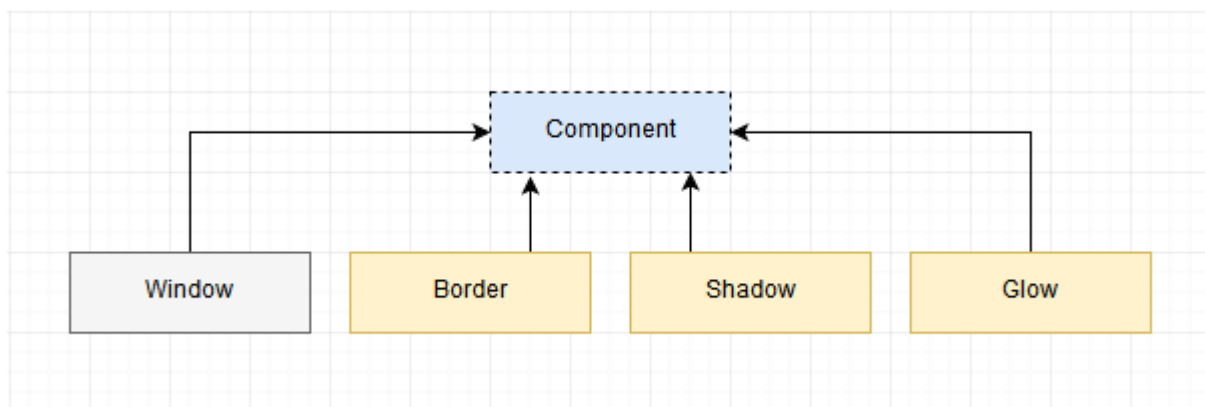
So, while it may seem that a vertical inheritance style would better fit our Window creation scenario there are considerable problems there as well. In the visual below, a Glow class will always be coupled to Shadow and Border whether you need them or not - for example, borders and shadows always come along for the ride even if you want to just apply a simple glow style.



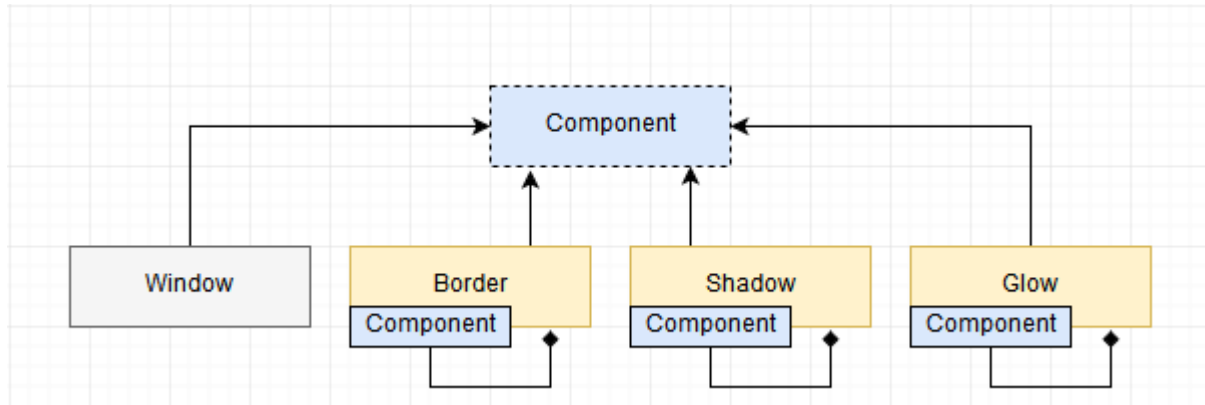
DECORATOR TO THE RESCUE

What you would really want is to make each inheriting class (Border, Shadow, Glow) ways of **'decorating'** the base class in any combination you'd want. That's to say you would want all objects to be *modular components* - like lego blocks - so that a Window could include both a Border and a Shadow or a Border and a Glow and so forth without writing complicated class structures to represent these possible combinations.

The key to solving this is to think of all of the classes as simply components that share a common class structure or operations. You can imagine they all inherit from an abstract class or interface called Component:

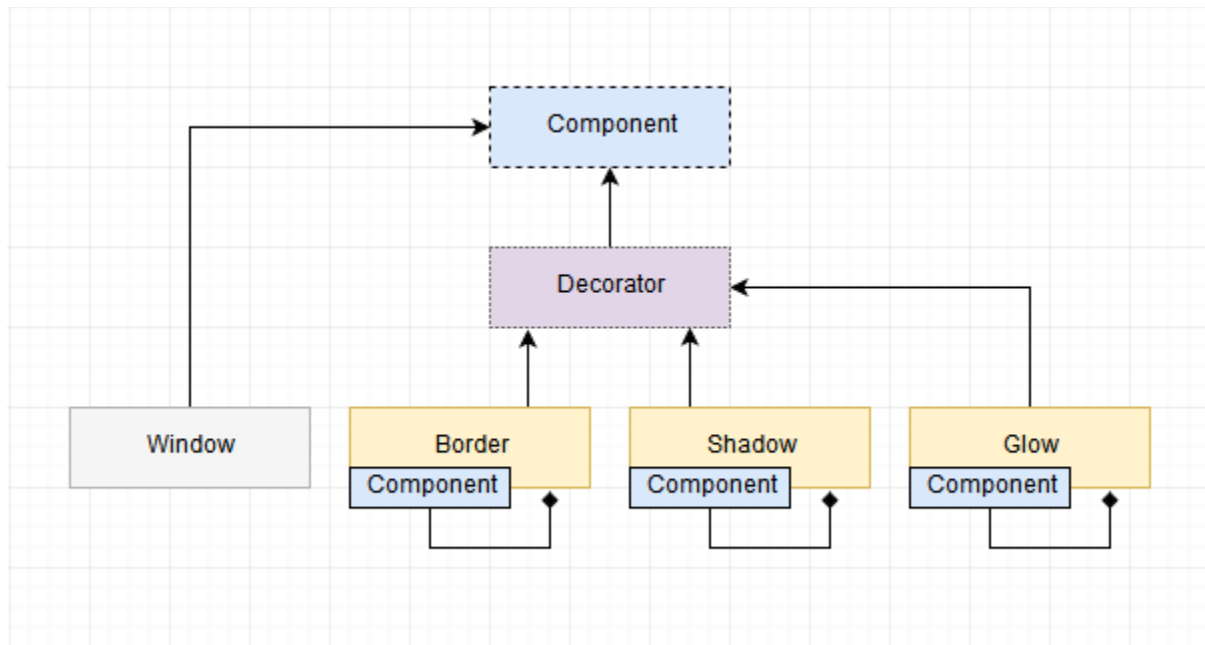


The next key is to consider that each Component can also hold a composition reference to another Component (represented by the diamond-tip arrow. For example: a Border can have a Shadow or Glow component):

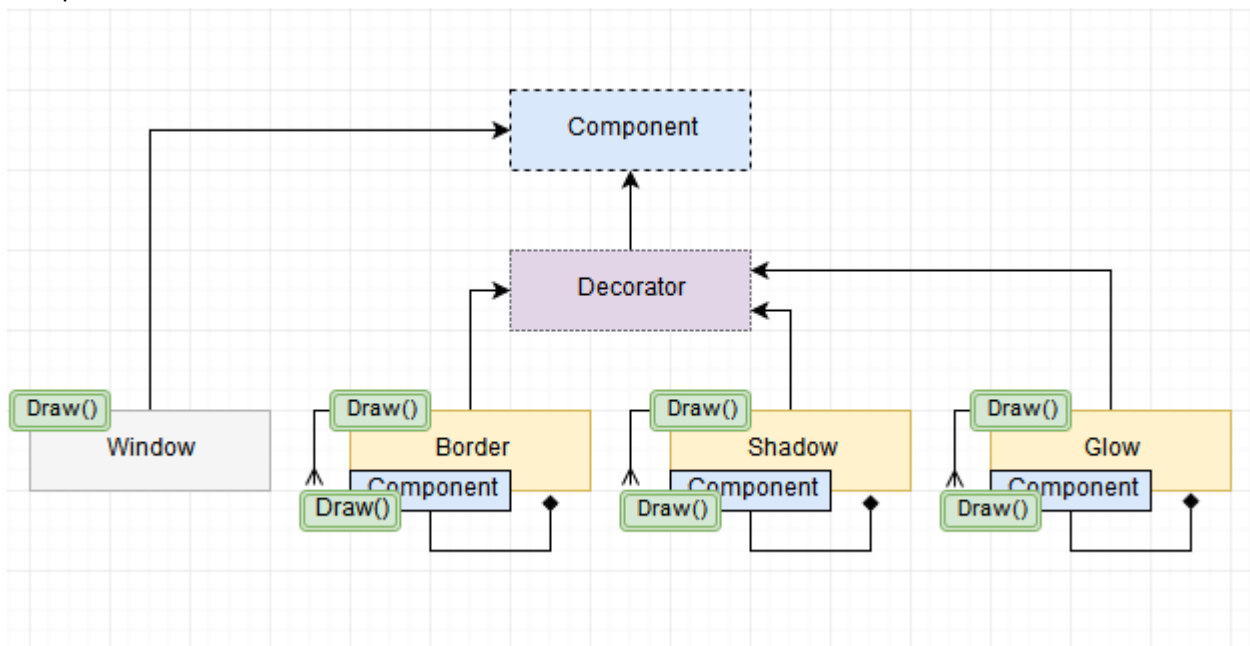


You can easily imagine how a Border object can contain a Glow or a Shadow object or any other possible combination and then terminate at a base Window object - essentially building up a single complex object by grafting other objects as you please using Russian Doll-like levels of composition. Since these objects all share a common Interface, we can imagine it just shares a common operation, let's say a **Draw()** method. So all you need is to be able to call each object's Draw() method in order to create the Window with it's arbitrary "decorations" at runtime. More on this in a moment.

Below is a minor modification which brings us closer to understanding the Decorator pattern. In the diagram below, the Border, Shadow and Glow classes inherit much of their basic functionality, as well as the component object reference, from a common Decorator subclass:



Now, let's add **Draw()** calls to our diagram showing each **Decorator** class's **Draw()** method call its **Component** field's **draw** method:



For example, say you have a **Glow** **Component** which has a **Border** **Component** field, and the **Border** **Component** has a **Window** **Component**. The **Glow.Draw()** method will call the **Border.Draw()** method which then calls the **Window.Draw()** method. In other words, each **Component** just has to call its **Draw()** method as well as its component field's **Draw()** method, creating a chain reaction that sums up to something resembling a complex single object.

DECORATOR SCAFFOLDING

Let's take this visual diagram and apply it to a scaffold representation of the Decorator pattern. We'll start by defining the fundamental Component interface which just makes sure all Components have a particular method. The **Operation()** method here is analogous to the Draw() method in our diagram.

```
//Basic interface for all Components
interface Component
{
    void Operation();
}
```

We also define well the simplest kind of Component (one which *doesn't* have another Component attached to it), which is called BaseComponent (analogous to the Window class in the diagram)

```
//Base Component - Doesn't hold a component reference
class BaseComponent : Component
{
    public void Operation()
    {
        Console.WriteLine("BaseComponent Operation()");
    }
}
```

You can think of the BaseComponent as the base lego block in that it can attach other lego blocks on top of it, but can't be attached to anything below it. You can also think of it as the most basic building block before it becomes "decorated" - kind of like a canvas of a painting.

Now we can define a Decorator as a type of Component which can have other Components attached to it via its Component field:

```
//Base Decorator - Can hold a component reference
abstract class Decorator : Component
{
    protected Component component;
    public void AttachComponent(Component component)
    {
        this.component = component;
    }
    public virtual void Operation()
    {
        if (component != null)
            component.Operation();
    }
}
```


Now you can define **concrete implementations of Decorators** that inherit from the abstract Decorator (analogous to the Border, Glow, Shadow classes in the diagram). Notice that these Decorators call their own Operation() method as well as the Operation() method of the Component attached to it (if attached):

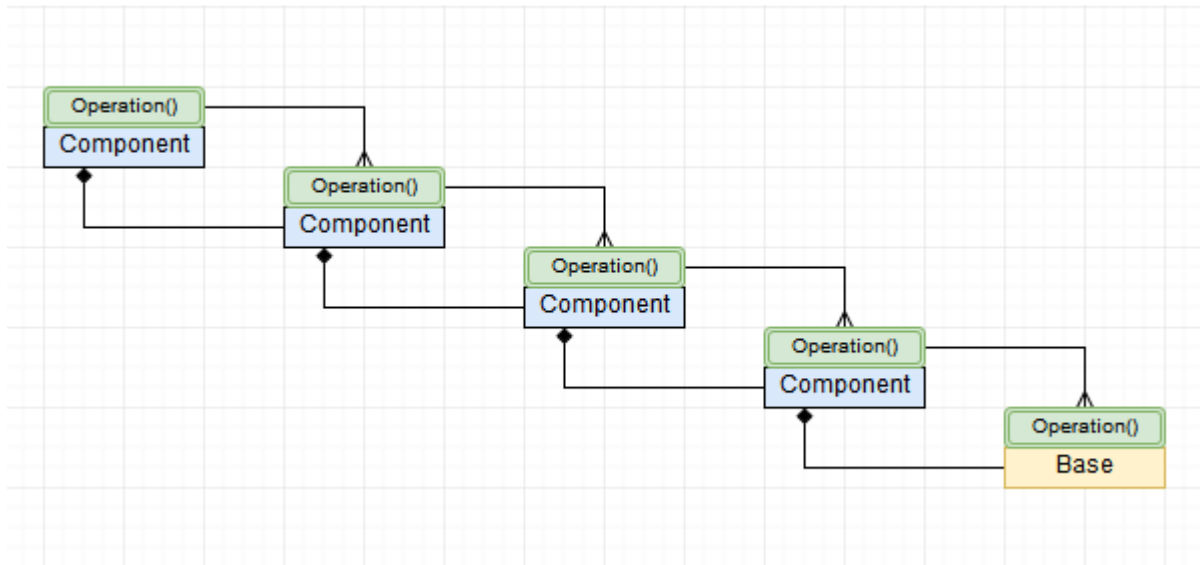
```
//Concrete Decorators - Can be added onto any other Decorator
class DecoratorA : Decorator
{
    public override void Operation()
    {
        base.Operation();
        Console.WriteLine("DecoratorA Operation()");
    }
}

class DecoratorB : Decorator
{
    public override void Operation()
    {
        base.Operation();
        Console.WriteLine("DecoratorB Operation()");
    }
}
```

And, finally, in a client class we can create the effect of building up a single object, just like lego blocks being attached together:

```
class Program
{
    static void Main()
    {
        BaseComponent basicComp = new BaseComponent();
        Decorator firstDecoration = new DecoratorA();
        firstDecoration.AttachComponent(basicComp);
        Decorator secondDecoration = new DecoratorB();
        secondDecoration.AttachComponent(firstDecoration);
        secondDecoration.Operation();
    }
}
```

The interesting thing here is the “secondDecoration” holds reference to the “firstDecoration” which in turn holds reference to the “basicComp” so when we call the secondDecoration.Operation() method it does all operations for all components, starting from the base and moving upwards. The following visual represents the call chain of component operations. This will be clearer when we step through the method calls using the debugger:



Let's now fully convert this scaffolding to resemble decorating a Window with different styles and better exemplifying the diagram above:

```

interface Component
{
    void Draw();
}

//Base Component
class Window : Component
{
    public void Draw()
    {
        Console.WriteLine("Drew a Window");
    }
}

//Base Decorator
abstract class Decorator : Component
{
    protected Component component;
    public void AttachComponent(Component component)
    {
        this.component = component;
    }
    public virtual void Draw()
    {
        if (component != null)
            component.Draw();
    }
}
  
```

```

//Concrete Style Decorators (Border, Shadow, Glow)
class Border : Decorator
{
    public override void Draw()
    {
        base.Draw();
        Console.WriteLine("Drew a Border Around the Window");
    }
}

class Shadow : Decorator
{
    public override void Draw()
    {
        base.Draw();
        Console.WriteLine("Drew a Shadow Around the Window");
    }
}

class Glow : Decorator
{
    public override void Draw()
    {
        base.Draw();
        Console.WriteLine("Drew a Glow Around the Window");
    }
}

```

We can now decorate the window in any way we choose because the decorators are modular components:

```

class Program
{
    static void Main()
    {
        Component window = new Window();
        Decorator border = new Border();
        Decorator shadow = new Shadow();
        Decorator glow = new Glow();

        glow.AttachComponent(window);
        border.AttachComponent(glow);
        border.Draw(); //draws window with glow, then border
    }
}

```

```

class Program
{
    static void Main()
    {
        ...
        border.AttachComponent(window);
        shadow.AttachComponent(border);
        shadow.Draw(); //draws window with border, then shadow
    }
}

```

To better understand the calling sequence for the Draw() methods, let's step through execution order by setting a breakpoint at the first Draw() call:



```

18 shadow.Draw(); //draws window with border, then shadow

```

When you step through the debugger with F11 you will see that we first enter through the Shadow's Draw() call:

```

class Shadow : Decorator
{
    public override void Draw()
    {
        base.Draw(); ≤ 1ms elapsed
        Console.WriteLine("Drew a Shadow Around the Window");
    }
}

```

It immediately goes to the base draw() method to see if a component is attached and run its Draw() method if it's attached:

```

//Base Decorator
abstract class Decorator : Component
{
    protected Component component;

    public void AttachComponent(Component component)
    {
        this.component = component;
    }

    public virtual void Draw()
    {
        if (component != null) ≤ 1ms elapsed
            component.Draw();
    }
}

```

Of course the Component for the Shadow object is attached and is a Border type of Component. The Border will then check to see if it has a Component is attached to it and repeats the same process:

```

//Concrete Style Decorators (Border, Shadow, Glow)
class Border : Decorator
{
    public override void Draw()
    {
        base.Draw(); ≤ 1ms elapsed
        Console.WriteLine("Drew a Border Around the Window");
    }
}

```

Since the Border's attached Component is a Window - which executes its Draw() method without checking for attached components (as a base component doesn't have any) - we finally get our first print out.

```
//Base Component
class Window : Component
{
    public void Draw()
    {
        Console.WriteLine("Drew a Window"); ≤ 1ms elapsed
    }
}
```

We then exit back to the next suspended Draw() call for the Border:

```
//Concrete Style Decorators (Border, Shadow, Glow)
class Border : Decorator
{
    public override void Draw()
    {
        base.Draw();
        Console.WriteLine("Drew a Border Around the Window"); ≤ 1ms elapsed
    }
}
```

And then the Shadow:

```
class Shadow : Decorator
{
    public override void Draw()
    {
        base.Draw();
        Console.WriteLine("Drew a Shadow Around the Window"); ≤ 1ms elapsed
    }
}
```

QUIZ QUESTIONS

Which best describes the role of a Decorator?

1. Ensuring every object is a component that can hold another component
2. Adding functionality to classes/objects by using an extensible composition scheme
3. Using a definite “vertical” inheritance to build indefinitely complex objects
4. None of the above

Inheritance can work well, but composition might be a better fit if:

1. You don’t need a common interface
2. Objects have very different functionality
3. inheritance leads to an explosion of hierarchical subclasses
4. None of the above

A Decorator tends to build hierarchies which are more

1. Horizontal
2. Vertical
3. Modular
4. Both A and C
5. Both B and C

A Decorator can be more flexible than Inheritance when:

1. Inheritance is mostly vertical
2. Objects can be looked at as components
3. Object functionality needs to be modular
4. None of the above

Which sounds most accurate for a real-world Decorator pattern using a Pizza as analogy?

1. All parts are Components
2. The shell is the Base Component
3. Toppings are Decorators
4. All of the above
5. None of the above

QUIZ ANSWERS

6. **(2) Adding functionality to classes/objects by using an extensible composition scheme**
7. **(3) inheritance leads to an explosion of hierarchical subclasses**
8. **(4) Both A and C**
9. **(3) Object functionality needs to be modular**
10. **(4) All of the above**

13. Composite Pattern

PATTERN TYPE: Structural

FREQUENCY OF USE: Medium-High

MEMORY HELPER: Object composition trees

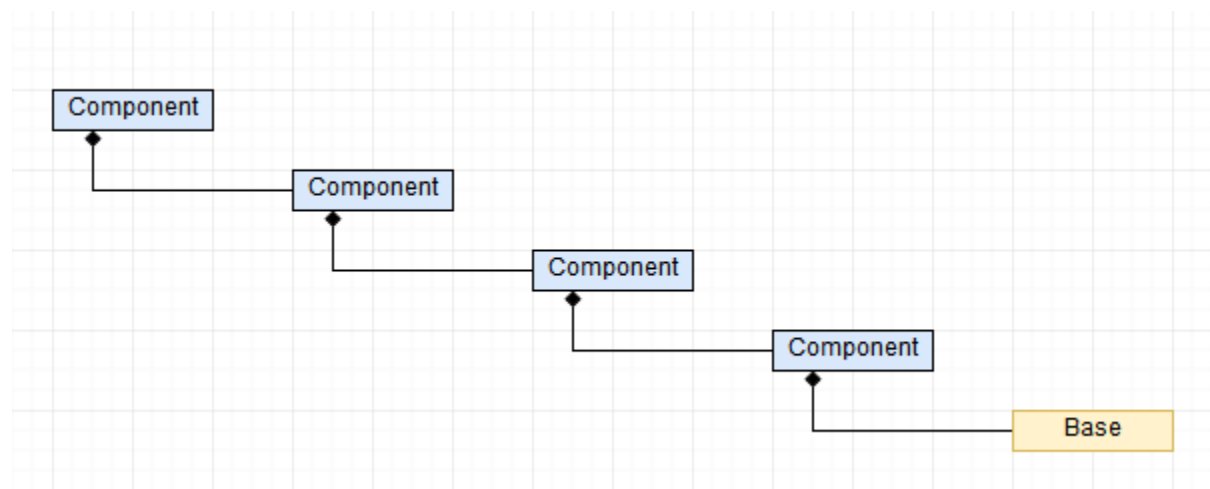
PATTERN OBJECTIVE: To create a tree-like composite of objects that can be treated as individual parts or as grouped composites.

COMPOSITE VS DECORATOR

The Composite pattern is very similar to the Decorator pattern in its general design, except for a few key differences of functionality and intent. The first key difference is how it chains components together.

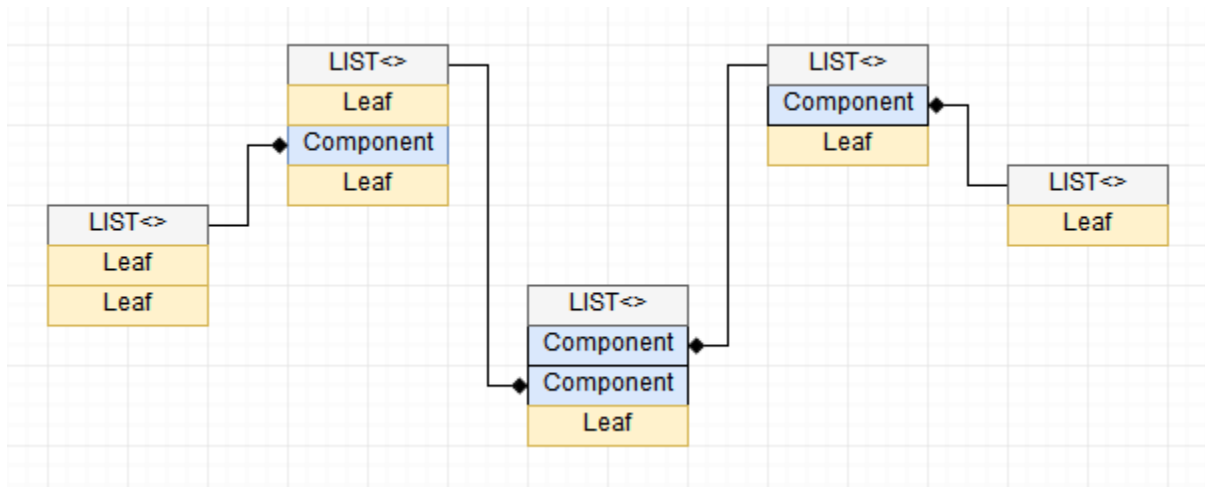
A **Decorator** is extremely basic in that each Decorator can hold a *single Component*, which in turn can hold another single Component. That chaining would look something like the diagram below, eventually terminating at a **Base** Component which can't hold any other Component reference:

DECORATOR BRANCHING



By contrast, a Composite Component holds a **List<Component>** which can itself hold an indefinite *series of Component references* that branch out in a very non-linear manner. Instead of having a single Base Component to terminate the branching, we have “**Leaf**” Components - which have the same basic role as a Base in that they cannot attach any additional Components, effectively terminating the branch:

COMPOSITE BRANCHING



It wouldn't be a stretch to say that the major difference between each approach could be reduced to the singular-vs-plural Component reference that forms the basis for these two patterns:

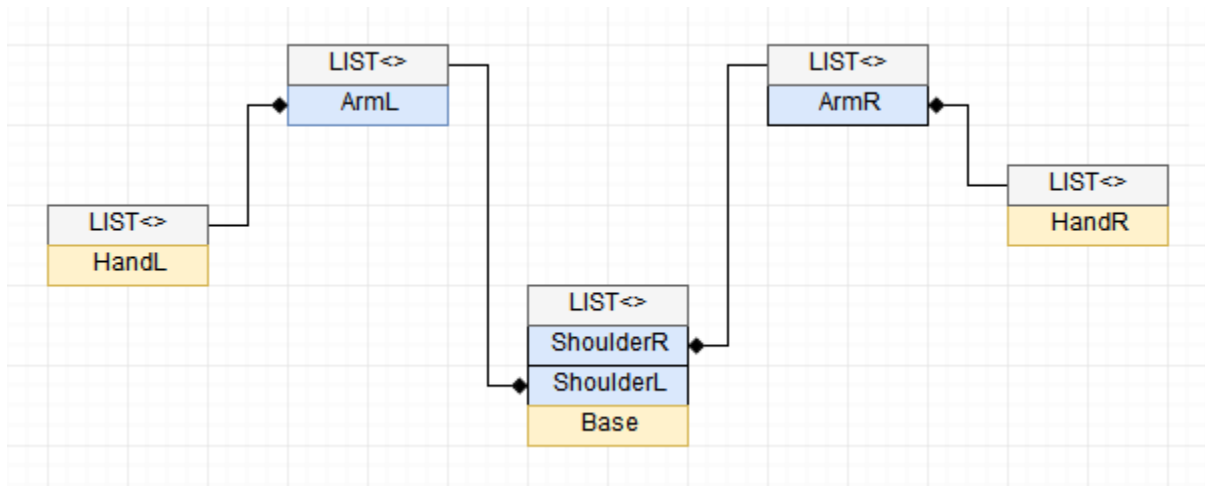
```
class Decorator : Component
{
    //Single Attached Component
    Component component;
}

class Composite : Component
{
    //List of Attached Components
    List<Component> components = new List<Component>();
}
```

In terms of intent, whereas a Decorator intends to add functionality block-by-block in order to build up functionality of a *single object*, a Composite is more geared towards representing a complex object that can be just as easily treated as *parts or as a whole*.

BUILDING A ROBOT (ANALOGY)

A real-world visual analogy is something like building a robot and separating it into grouped components. In this way you can easily separate the body, hands and arms using a tree-like composite structure:



If you wanted to add fingers to this design, you would just make HandL and HandR an adaptable Component rather than a terminating Leaf and then attach five Leafs on each Hand to represent the fingers.

Recalling that a Decorator chains **Operation()** methods to create a single unified behaviour, you can imagine the same holding true with a Composite. Using the diagram above as analogy, you can imagine that an Operation() call from a Shoulder sends the Operation() call down the chain to the Arm and from the Arm to the Hand, etc in order to form a complex operation that effectively unifies all of those elements in the chain.

COMPOSITE SCAFFOLDING

Let's take this visual analogy of building a robot via separate, yet grouped, components and apply it to a scaffold representation of the Composite pattern. We'll start by defining our three main classes:

```

abstract class Component
{
}

class Composite : Component
{
    private List<Component> subComps = new List<Component>();
}

class Leaf : Component
{
}
  
```

Next we'll add a Constructor to the base Component class which will also become the default Constructor for our Composites and Leafs:

```
abstract class Component
{
    protected string PartName;

    public Component(string partName)
    {
        PartName = partName;
    }
}

class Composite : Component
{
    private List<Component> subComps = new List<Component>();

    public Composite(string partName) : base(partName) { }
}

class Leaf : Component
{
    public Composite(string partName) : base(partName) { }
}
```

Next we'll require all Components to have Add() and Remove() methods by marking them as abstract in the base Component class:

```
abstract class Component
{
    protected string PartName;
    public Component(string partName)
    {
        PartName = partName;
    }

    public abstract void Add(Component component);
    public abstract void Remove(Component component);
}
```

We'll implement a basic **List<>** Add/Remove routine for Composites while Leaf's will simply throw an Exception since they do not contain Component references:

```

class Composite : Component
{
    private List<Component> subComps = new List<Component>();
    ...
    public override void Add(Component component)
    {
        subComps.Add(component);
    }

    public override void Remove(Component component)
    {
        subComps.Remove(component);
    }
}

class Leaf : Component
{
    ...
    public override void Add(Component component)
    {
        throw new Exception("Attaching Component to Leaf Not Allowed!");
    }
    public override void Remove(Component component)
    {
        throw new Exception("Removing Component from Leaf Not Allowed!");
    }
}

```

Now to traverse the connections between all of the Components, we'll simulate a "wiring" process by first defining its default implementation in the base class. This is pretty much the same principle behind the recursive Operation() method in a Decorator. The "length" and "gauge" parameters here will just add a bit of visual flare to the displayed output:

```

abstract class Component
{
    ...
    public virtual void WireUp(int length, string gauge)
    {
        string wire = "";
        for (int i = 1; i < length; i++)
            wire += " ";

        wire += gauge;
        Console.WriteLine(wire + PartName);
    }
}

```

The Composite will call the base **WireUp()** implementation and then iterate through its attached Components - calling the same WireUp() process (but lengthening the “wire” on every containment level so as to visually emphasizes the tree-like appearance):

```
class Composite : Component
{
    ...
    public override void WireUp(int length, string gauge)
    {
        base.WireUp(length, gauge);

        foreach (Component component in subComps)
            component.WireUp(length + 1, gauge);
    }
}
```

Leafs, of course, don’t have any attached Components so they simply use the the base WireUp() implementation (there’s no sense overriding it, but we’ll do so here to be explicit):

```
class Leaf : Component
{
    ...
    public override void WireUp(int length, string gauge)
    {
        base.WireUp(length, gauge);
    }
}
```

Now all that’s left to do is construct and wire up our “robot” to see how its components connect to one another. We start by creating a right “hand” - which we know will have to be a Composite as it will contain “fingers.” We can add the fingers via a simple **for()** loop and attach that entire Composite (hand with fingers) to a “shoulder” Composite:

```
class Program
{
    static void Main()
    {
        //Right Parts of Robot
        Component handR = new Composite("Right Hand");
        for (int i=1; i <= 5; i++)
            handR.Add(new Leaf("Finger" + i));

        Component armR = new Composite("Right Arm");
        armR.Add(handR);
        Component shoulderR = new Composite("Right Shoulder");
        shoulderR.Add(armR);
    }
}
```

We then repeat this process for the “left” side of the robot:

```
//Left Parts of Robot
Component handL = new Composite("Left Hand");
for (int i = 1; i <= 5; i++)
    handL.Add(new Leaf("Finger" + i));

Component armL = new Composite("Left Arm");
armL.Add(handL);
Component shoulderL = new Composite("Left Shoulder");
shoulderL.Add(armL);
```

We then attach both sides to a “body” Composite, as well as a “base” which can contain no further Components, and is therefore a Leaf:

```
//Attach Left/Right Parts to Body
Component body = new Composite("Body");
body.Add(shoulderR);
body.Add(shoulderL);

Component Base = new Leaf("Base");
body.Add(Base);
```

When we run the WireUp() method from the Body it will show us a visual representation of the entire tree-like connections starting from the Body:

```
//Wire Up to View Connections
body.WireUp(1, "!=");
```

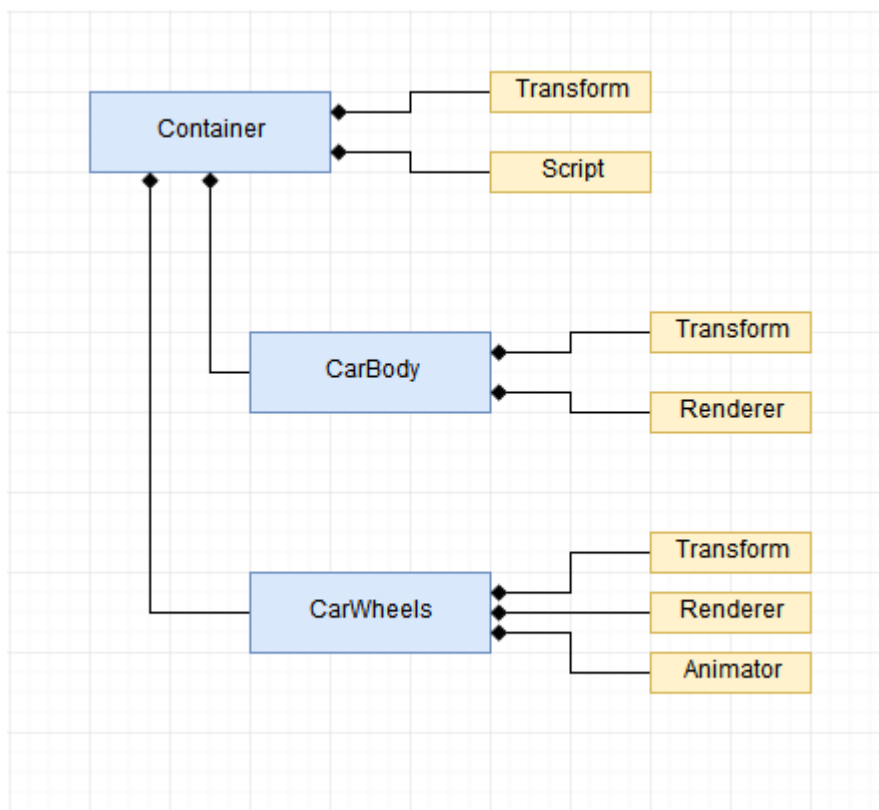
```
==Body
==Right Shoulder
==Right Arm
==Right Hand
==Finger1
==Finger2
==Finger3
==Finger4
==Finger5
==Left Shoulder
==Left Arm
==Right Hand
==Finger1
==Finger2
==Finger3
==Finger4
==Finger5
==Base
```

COMPONENT BASED GAME ENGINE EXAMPLE

For our real-world example we're going to simulate the way a component-based game engine groups components to create whole structures within its game world. The technique we're replicating is very similar to how something like the Unity Engine works, so you can keep that in mind if you're familiar with that Framework.

Our mock system will have GameObjects acting as Composites that contain other GameObjects or Leaf Components. For example, we'll have a unified **Car** GameObject while keeping the **Wheels** and **Body** as separate GameObjects. This allows us to attach Components as we please to implement different transform positions, rendered visuals, animations and scripts for the different GameObjects.

We will then attach the Body and Wheels Composites to a parent Composite GameObject, allowing us to treat the entire Composite structure as if it were a single object. The entire connected structure can be visualized as follows (blue boxes indicating Composite GameObjects and yellow boxes indicating attached Leaf Components):



Let's start our project by defining the base Component class. All Components will also contain references to their **Parent** GameObjects:


```
//Component
abstract class Component
{
    protected string name;
    public GameObject Parent;
    public Component(string name)
    {
        this.name = name;
    }
    public abstract void Add(Component component);
    public abstract void Remove(Component component);
}
```

We then define the **Composite** GameObject class. Note how the **Add()** method attaches a Component to the GameObject while also giving that Component a reference to the containing Parent GameObject that attached it. This is so an attached Component can know which GameObject it's attached to, which will be important later on:

```
//Composite
class GameObject : Component
{
    public GameObject(string name) : base(name) { }
    public List<Component> subComps = new List<Component>();

    public override void Add(Component component)
    {
        component.Parent = this;
        subComps.Add(component);
    }

    public override void Remove(Component component)
    {
        subComps.Remove(component);
    }
}
```

Next, we'll define a base class for a **Leaf** that all Leaf Components will inherit implementation from:

```
//Abstract Leaf
abstract class Leaf : Component
{
    public Leaf(string name) : base(name) { }

    public override void Add(Component component)
    {
        throw new Exception(name + " is a Leaf. Cannot Add Component.");
    }

    public override void Remove(Component component)
    {
        throw new Exception(name + " is a Leaf. Cannot Remove Component.");
    }
}
```

Since we won't be adding any particular implementation to specific Leaf Components, we'll just broadly define the various types:

```
//Concrete Leafs
class Transform : Leaf
{
    public Transform(string name) : base(name) { }
}

class Renderer : Leaf
{
    public Renderer(string name) : base(name) { }
}

class Animator : Leaf
{
    public Animator(string name) : base(name) { }
}

class Script : Leaf
{
    public Script(string name) : base(name) { }
}
```

Next, we'll want to trace connections between GameObject's and their attached Components by iterating through the tree of connections. We can add this default implementation to the base Component class by assuming the Component object is a GameObject (Composite):

```

abstract class Component
{
    ...
    public virtual void TraceFwd()
    {
        Console.WriteLine("\n" + name + " (Composite) has Connected to it:");
        GameObject g = this as GameObject;
        foreach (Component c in g.subComps)
            c.TraceFwd();
    }
}

```

And since this method is marked **virtual**, we can override the Implementation in the case the Component is a Leaf:

```

abstract class Leaf : Component
{
    ...
    public override void TraceFwd()
    {
        Console.WriteLine(name + " (Leaf)");
    }
}

```

We'll also create an override if the Leaf is a Script to allow for a special method to be run (in this case the "Update Loop"):

```

class Script : Leaf
{
    ...
    public override void TraceFwd()
    {
        base.TraceFwd();
        Update();
    }

    private void Update()
    {
        Console.WriteLine(" [ Update Loop Running ] ");
    }
}

```

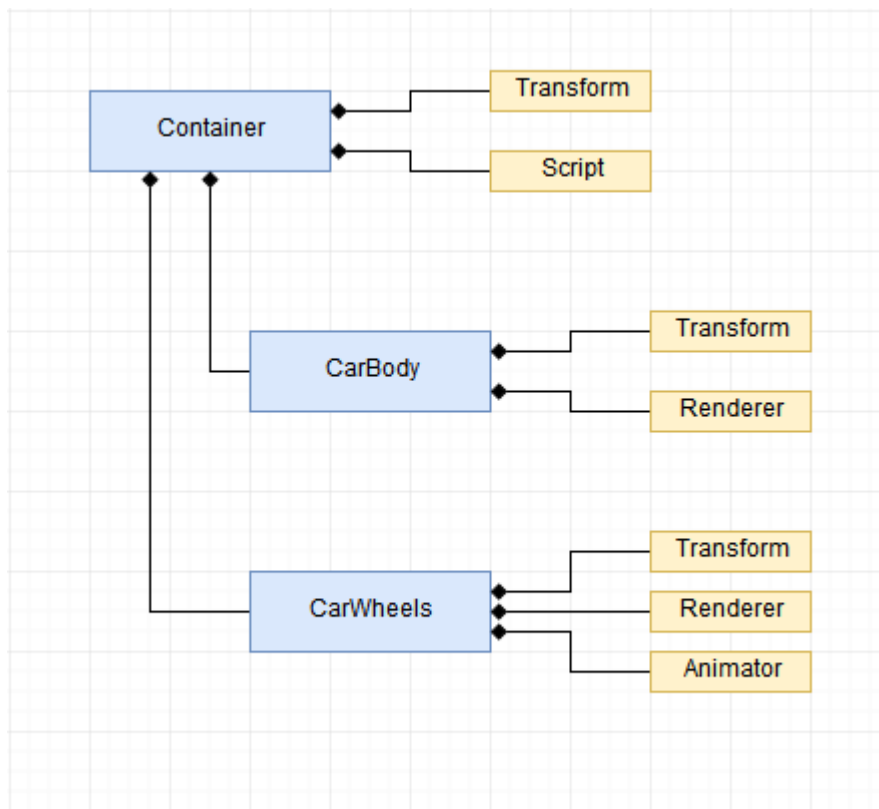
Now that we've got the ability to trace connections *forward* - just as we had with the scaffold example - let's also add the ability to trace connections *backward* by tapping into the **Parent** reference for each Component. Again, we can just define this default implementation in the base Component class:

```

abstract class Component
{
    ...
    public virtual void TraceBack()
    {
        Console.Write(this.name);
        if (Parent != null)
        {
            Console.Write(" Traces Back to ");
            Parent.TraceBack();
        }
        Console.WriteLine();
    }
}

```

Let's bring this all together to form the connection tree we saw earlier:



We start by creating the three main GameObject's and adding their Components one-by-one:

```

class Program
{
    static void Main()
    {
        GameObject Go1 = new GameObject("Container");
        Component T1 = new Transform("ContainerTransform");
        Component S1 = new Script("ContainerScript");
        Go1.Add(T1);
        Go1.Add(S1);
        GameObject Go2 = new GameObject("CarBody");
        Component T2 = new Transform("BodyTransform");
        Component R1 = new Renderer("BodyRenderer");
        Go2.Add(T2);
        Go2.Add(R1);
        GameObject Go3 = new GameObject("CarWheels");
        Component T3 = new Transform("WheelsTransform");
        Component R2 = new Renderer("WheelsRenderer");
        Component A1 = new Animator("WheelsAnimator");
        Go3.Add(T3);
        Go3.Add(R2);
        Go3.Add(A1);
    }
}

```

Since the main container GameObject (Go1) acts as the parent to the two other GameObjects, we attach those at the very end:

```

Go1.Add(Go2);
Go1.Add(Go3);

```

Here are the outputs for tracing the connections forward, plus a visual diagram:

```

static void Main()
{
    ...
    Go1.TraceFwd();
}

```

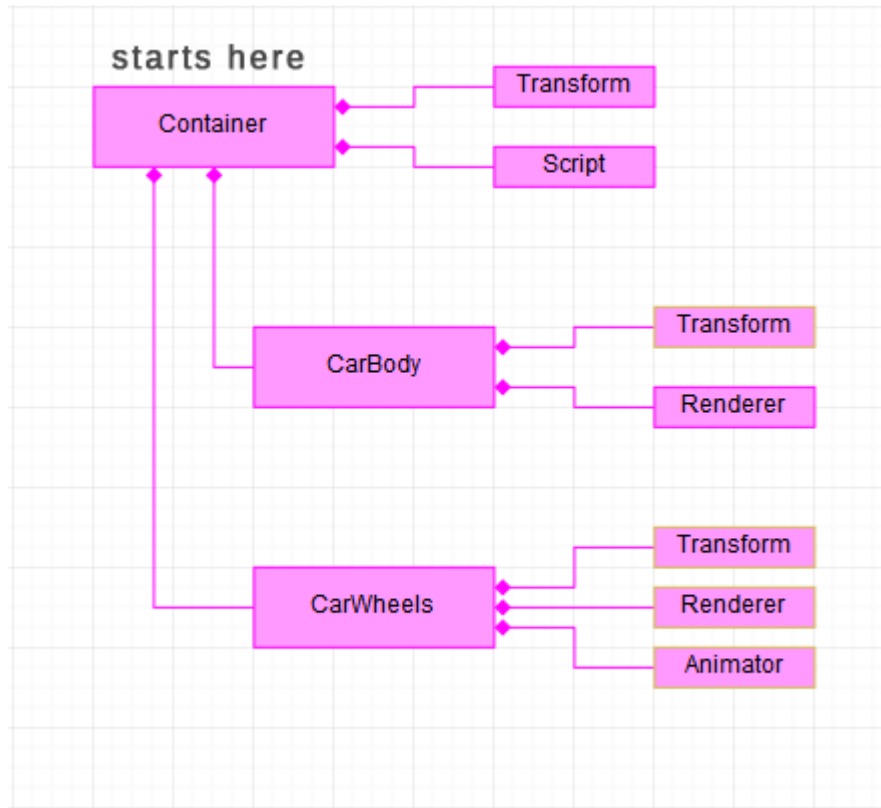
```

Container <Composite> has Connected to it:
ContainerTransform <Leaf>
ContainerScript <Leaf>
[ Update Loop Running ]

CarBody <Composite> has Connected to it:
BodyTransform <Leaf>
BodyRenderer <Leaf>

CarWheels <Composite> has Connected to it:
WheelsTransform <Leaf>
WheelsRenderer <Leaf>
WheelsAnimator <Leaf>

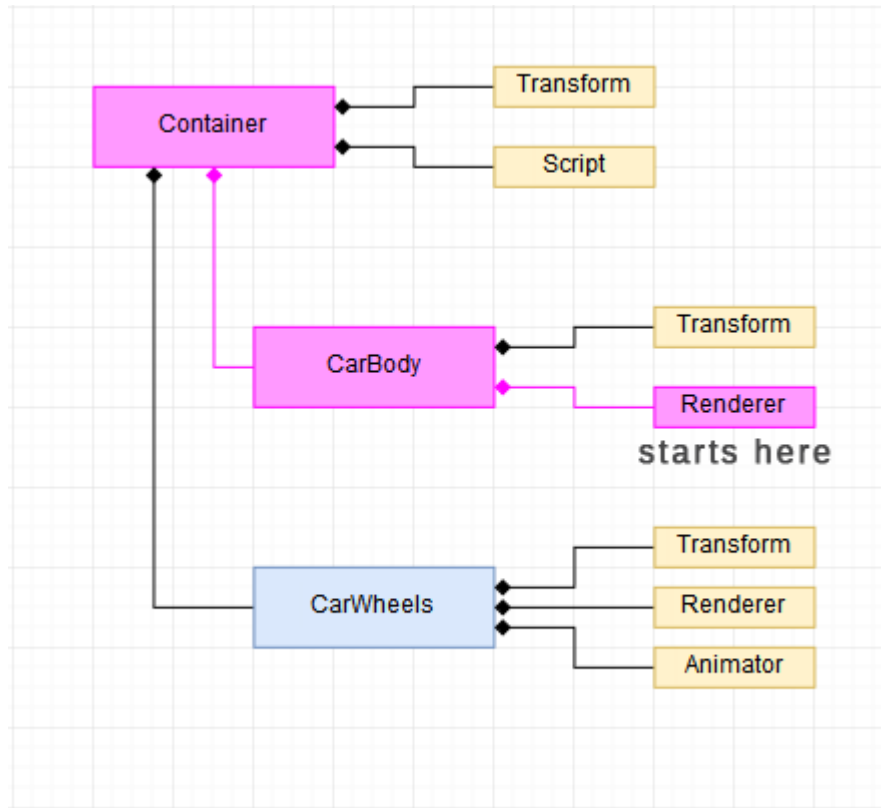
```



Here are the outputs for tracing the connections backward, plus a visual diagram:

```
static void Main()
{
    ...
    R1.TraceBack();
```

BodyRenderer Traces Back to CarBody Traces Back to Container



QUIZ QUESTIONS

Which of the following is a disadvantage of a Composite?

1. Leafs can't know about their parent Composites
2. All Components are treated uniformly
3. Leafs need to rely on runtime checks to restrict behaviour
4. None of the above

Inheritance can work well, but composition might be a better fit if:

1. You don't need a common interface
2. Objects have very different functionality
3. Inheritance leads to an explosion of hierarchical subclasses
4. None of the above

A Decorator tends to build hierarchies which are more

1. Horizontal
2. Vertical
3. Modular
4. Both A and C
5. Both B and C

A Decorator can be more flexible than Inheritance when:

1. Inheritance is mostly vertical
2. Objects can be looked at as components
3. Object functionality needs to be modular
4. None of the above

Which sounds most accurate for a real-world Decorator pattern using a Pizza as analogy?

1. All parts are Components
2. The shell is the Base Component
3. Toppings are Decorators
4. All of the above
5. None of the above

QUIZ ANSWERS

1. **(3) Leafs need to rely on runtime checks to restrict behaviour**
2. **(3) Inheritance leads to an explosion of hierarchical subclasses**
3. **(4) Both A and C**
4. **(3) Object functionality needs to be modular**
5. **(4) All of the above**

14. Template Method Pattern

PATTERN TYPE: Behavioral

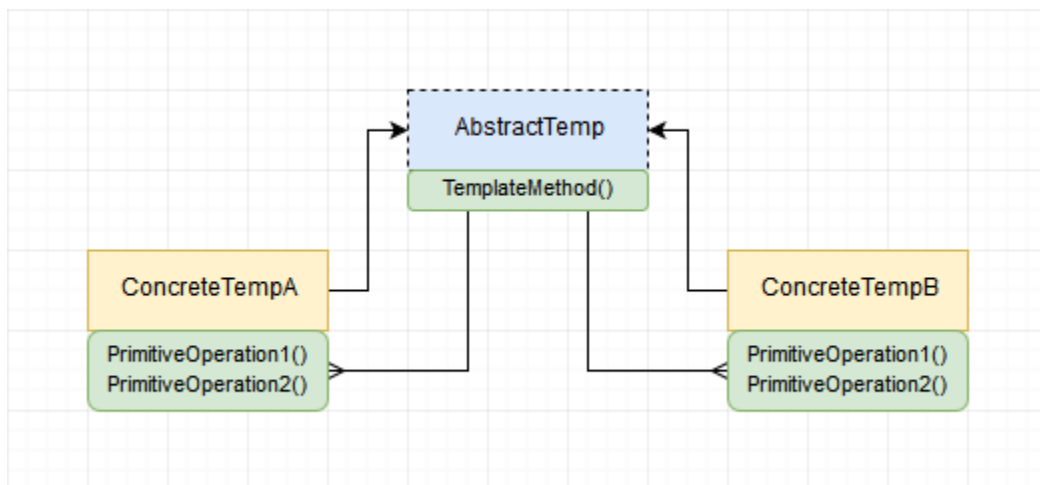
FREQUENCY OF USE: Medium

MEMORY HELPER: Default method call sequence

PATTERN OBJECTIVE: To specify a method that calls a common sequence of sub-methods, while allowing subclasses to redefine some parts of the sequence.

TEMPLATE METHOD SCAFFOLD

The Template Method pattern is very simple to understand. The most basic formulation of it starts with an abstract class that contains a non-abstract method (marked as the TemplateMethod below) which then calls a sequence of sub-methods. Concrete versions inherit from this abstract class and contain the method details for the sub-methods (marked below as PrimitiveOperation1/PrimitiveOperation2). Here is how you can visualize that basic structure:



Here is what the code looks like, starting with the AbstractTemp class:

```
abstract class AbstractTemp
{
    //The "Template Method"
    public void Sequence()
    {
        PrimitiveOperation1();
        PrimitiveOperation2();
    }

    public abstract void PrimitiveOperation1();
    public abstract void PrimitiveOperation2();
}
```

And now the sub-methods, along with their unique details, defined in any number of Concrete classes that derive from AbstractTemp:

```
class ConcreteTempA : AbstractTemp
{
    public override void PrimitiveOperation1()
    {
        Console.WriteLine("ConcreteTempA's PrimitiveOperation1 Method");
    }

    public override void PrimitiveOperation2()
    {
        Console.WriteLine("ConcreteTempA's PrimitiveOperation2 Method");
    }
}

class ConcreteTempB : AbstractTemp
{
    public override void PrimitiveOperation1()
    {
        Console.WriteLine("ConcreteTempB's PrimitiveOperation1 Method");
    }

    public override void PrimitiveOperation2()
    {
        Console.WriteLine("ConcreteTempB's PrimitiveOperation2 Method");
    }
}
```

In the client, we get different sequence of behaviors depending on the subclass we instantiate:

```
class Program
{
    static void Main()
    {
        AbstractTemp a = new ConcreteTempA();
        a.Sequence();

        Console.WriteLine();

        AbstractTemp b = new ConcreteTempB();
        a.Sequence();
    }
}
```

INVERSION OF CONTROL

While many inheritance scenarios involve child subclasses calling base methods, the Template Method pattern inverts this typical process. Basically the parent class tells the children which of their methods it expects to call. In effect, the parent class is imposing behavioral structure on the child subclasses which can help subclasses keep design principles such as reduced repetition, single responsibility and simplified, predictable structure. This is desirable, in part, because the inverse - letting children optionally extend and call parent behavior - is far less obvious and predictable when adding subclasses.

WHAT'S THE POINT?

While the basic formulation we looked at has shown us a general way of imposing design principles, it doesn't really hint at why this pattern has any *particular* usefulness. At first glance, it might simply appear to be a behavioural version of the Factory Method and Facade patterns, with little reason to use it instead. The key to understanding its unique utility is in considering its typical application scenario where there is a sequence of operations that several subclasses might implement. Think of these abstract, yet common sequences:

- **Open/Write/Close**
- **Open/Read/Close**
- **Connect/Process/Disconnect**
- **Start/Run/End**

The first three in the list might be file operations, database query operations, or possibly even network connections. The last one might represent a typical program launcher or hub application. In all cases, these are common sequences where the beginning and end operations are invariant while there might be some variance in the middle operation(s). In other words, Write/Read/Process/Run operations will be different depending on the subclass operation being done while Open/Close/Connect/Disconnect/Start/End are invariant routines.

Since the beginning and end parts won't change, and since we expect this all to happen in a single sequence, we can "template" away those operations while letting subclasses redefine how it handles the middle "primitive" operations.

DATABASE QUERY TEMPLATE

Take, for example, a typical query scenario for retrieving information from a database. Let's say we have a database that contains a table for products and another table for written articles.

Whether a query returns a product or an article - two very different data results - we expect the connect/disconnect process to be the same. However, the "select" criteria as well as other processes to be done on the resulting data would probably be quite different.

What we can do is template the entire process, while letting the Product and Article subclasses redefine the select/process behavior. Here is what that would look like using the Template Method:

```
class Program
{
    static void Main()
    {
        /* Gets Brooms in Product Inventory */
        AbstractDb DbQuery1 = new Product();
        DbQuery1.RunSequence("Broom");
        /* Gets Articles on Dust */
        AbstractDb DbQuery2 = new Article();
        DbQuery2.RunSequence("Dust");
    }
}

abstract class AbstractDb
{
    //The "Template method"
    public void RunSequence(string constraint)
    {
        Connect();
        Select(constraint);
        Disconnect();
    }
    public abstract void Select(string constraint);

    public void Connect()
    {
        Console.WriteLine("Connecting To Database");
    }
    public void Disconnect()
    {
        Console.WriteLine("Disconnecting from Database\n");
    }
}
```

```

class Product : AbstractDb
{
    public override void Select(string productName)
    {
        Console.WriteLine("SELECT * FROM products WHERE name = " + productName);
    }
}

class Article : AbstractDb
{
    public override void Select(string articleName)
    {
        Console.WriteLine("SELECT * FROM articles WHERE name = " + articleName);
    }
}

```

FRAMEWORKS AND EXTENSIBILITY

The real advantage of using this pattern is the simplicity of adding additional concrete subclasses that use the template in question. If our code example represented a *framework* for handling database queries, we've effectively created a system that lets users redefine only the parts that they are concerned with and not worry about defining common background functionality. For this reason, it's fairly common to see this pattern in Frameworks that expose a templated way of adding behaviors.

QUIZ QUESTIONS

Which best describes the objective of a Template Method?

1. Define invariant behavior once, while letting subclasses define variant behavior
2. Keep invariant behavior hidden, while exposing variant behavior
3. Abstract away variant behavior
4. None of the above

Which of the following is NOT a good example of a Template Method type of sequence?

1. Car Start/Drive/Stop sequence
2. Computer's Startup/Run-OS/Shutdown sequence
3. Word processor's Create/Edit/Save sequence
4. None of the above

A Template Method tries to affect inheritance by:

1. Making it more rigid
2. Making it more flexible
3. Limiting amount of subclasses
4. Limiting behavior

A Template Method is different from Factory Method in that:

1. Template Method is only useful in Frameworks
2. Factory Method isn't good for extensibility
3. Template Method makes subclassing easier, Factory Method doesn't
4. Factory Method creates objects, Template Method defines behaviors

QUIZ ANSWERS

1. **(1) Define invariant behavior once, while letting subclasses define variant behavior**
2. **(4) None of the above**
3. **(1) Making it more rigid**
4. **(4) Factory Method creates objects, Template Method defines behaviors**

15. State Pattern

PATTERN TYPE: Behavioral

FREQUENCY OF USE: Medium-High

MEMORY HELPER: Change object behavior

PATTERN OBJECTIVE: Delegate an object's behavior to an internal composition object, allowing behaviors to be swapped out seamlessly. This can have the appearance of the object changing its type at runtime.

HOW TO THINK OF "STATE"

Obviously the nature of most objects is to hold ever-changing state, typically through its fields and properties that have ever-changing values. But what if the object needs its methods to change, kicking off a different process depending on what "state" it's in?

For example, an application that tracks membership accounts of any kind might have members in at least one of two possible states: active and suspended. And depending on that member's state, it can change the implementation details of what that member can do - branching-off implementation details depending on a particular condition that the object is in. You might imagine solving this using code that looks something like below:

```
enum AccessState
{
    Platinum,
    Gold,
    Silver,
    Bronze
}

class Member
{
    AccessState access;
    public void Download(string urlRequest)
    {
        if (access == AccessState.Platinum || access == AccessState.Gold)
            Console.WriteLine("Downloading requested file: " + urlRequest);
        else
            Console.WriteLine("Download Denied for requested file: " + urlRequest);
    }
}
```

```

class Program
{
    static void Main()
    {
        Member member = new Member();
        member.AccessState = AccessState.Gold;
        member.Download("product253.zip");
    }
}

```

This solution uses a simple **enum** to connote a particular “state” the object is in, and then rely on conditionals in the methods referencing the state to change behavior depending on that state.

While this approach might seem reasonable given the simplicity shown, in a more complex real world scenario this will likely result in multiple conditional statements and monolithic methods. Even worse, adding to the system later (with additional conditionals/states, for instance) could become quite difficult.

Using a State pattern to solve this problem replaces the enum with actual objects that end up handling the variable implementation details. So you could imagine having Platinum, Gold, Silver and Bronze “State” objects with their own different Download() methods. These objects will also be able to change to a different State object if necessary.

STATE PATTERN SCAFFOLD

Let’s take a look at the simplest possible implementation of the State pattern. First, we’ll define a simple interface that all concrete States inherit from so that we can polymorphically change the state object by taking in the “Subject” that contains it (note: the GoF calls this class the “Context” but we’re using the term “Subject” instead as it carries greater meaning).

```

/* States */
interface IState
{
    void Switch(Subject subject);
}

```

Then, we'll define with the **Subject** class which contains the **State** object we end up passing into the `Switch()` method defined in the interface:

```
/* Subject */
class Subject
{
    public IState State;

    public Subject(IState state)
    {
        State = state;
    }

    public void StateChange()
    {
        State.Switch(this);
    }
}
```

Next, we can add any number of concrete States with their own specific implementation details by inheriting from the given interface:

```
class ConcreteStateA : IState
{
    public void Switch(Subject subject)
    {
        subject.State = new ConcreteStateB();
        Console.WriteLine(subject.State.GetType().Name);
    }
}

class ConcreteStateB : IState
{
    public void Switch(Subject subject)
    {
        subject.State = new ConcreteStateA();
        Console.WriteLine(subject.State.GetType().Name);
    }
}
```

In a client we can create the Subject object and change its state by calling the `StateChange()` method. In a real scenario the State object would probably contain completely different implementation details, having the effect of changing the Subject's behavior.

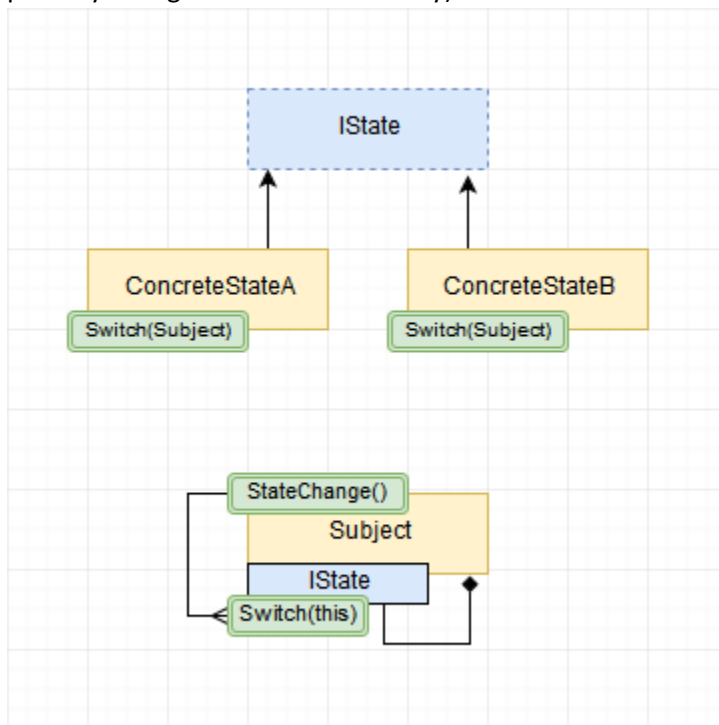
```

class Program
{
    static void Main()
    {
        Subject subject = new Subject(new ConcreteStateA());
        subject.StateChange(); //ConcreteStateB
        subject.StateChange(); //ConcreteStateA
        subject.StateChange(); //ConcreteStateB
        subject.StateChange(); //ConcreteStateA
    }
}

```

One of the interesting features of this pattern is that both the Subject and its internal State object have a bidirectional association. In this example, we saw that the Subject holds dependency to the State via Composition, while a State has a dependency to the Subject in the Switch() method parameter.

You can visualize the associations as being something like the diagram below. This shows how the Subject contains the State, the Subject then feeds *itself* into the Switch() method which sends the Subject's reference over to the State's Switch() method, allowing the State to modify the Subject (and possibly change its State if necessary):



STATE PATTERN ANALOGY

We're going to use a Chemistry analogy to hopefully drive the point home of how to think about the problem that a State pattern can solve.

We'll start by defining a simple interface defining the public methods for a State and Subject respectively:

```
interface IState
{
    void StateCheck(ICompound material);
    void Draw(ICompound material);
}

interface ICompound
{
    int Temp { get; set; }
    IState State { get; set; }
    void Draw();
}
```

Next, we'll define one possible Subject that shares this interface:

```
class Water : ICompound
{
    public Water(IState state, int temp)
    {
        this.State = state;
        this.Temp = temp;
    }
    public IState State { get; set; }
    private int _temp;
    public int Temp
    {
        get
        {
            return _temp;
        }
        set
        {
            _temp = value;
            StateCheck();
        }
    }
    private void StateCheck()
    {
        State.StateCheck(this);
    }
    public void Draw()
    {
        State.Draw(this);
    }
}
```

We see from this definition that this type of ICompound - Water - has State and Temp (temperature) properties. When the Temp value changes, the property sends the Water object to the State's StateCheck() method, where it might change to one of several different States depending on the Water object's temperature. Let's now define those three basic possible states:

```
class Liquid : IState
{
    public void StateCheck(ICompound material)
    {
        if (material.Temp <= 0)
            material.State = new Solid();
        else if (material.Temp >= 100)
            material.State = new Gas();
    }
    public void Draw(ICompound material)
    {
        Console.WriteLine("/|/|/|/|/|/|/|/|");
    }
}

class Solid : IState
{
    public void StateCheck(ICompound material)
    {
        if (material.Temp >= 100)
            material.State = new Gas();
        else if (material.Temp > 0)
            material.State = new Liquid();
    }
    public void Draw(ICompound material)
    {
        Console.WriteLine("[|][|][|][|][|][|][|][|]");
    }
}

class Gas : IState
{
    public void StateCheck(ICompound material)
    {
        if (material.Temp <= 0)
            material.State = new Solid();
        else if (material.Temp < 100)
            material.State = new Liquid();
    }
    public void Draw(ICompound material)
    {
        Console.WriteLine("* * * * *\n * * * \n  *");
    }
}
```

In the client, we can now make a Water object and see its State change as we change its temperature:

```
class Program
{
    static void Main()
    {
        ICompound water = new Water(new Liquid(), 100);
        water.Draw();
    }
}
```

WHO CONTROLS STATE TRANSITIONS?

You will note that each possible State contains the logic for changing into one of the other two States, and that *much of that logic is redundant*. Of course, we certainly could have made the StateCheck() only in the Subject (Water) and allow the Subject to change *its* State. It might be a good exercise to try this out for yourself.

This specific question of State-changing responsibility is a central concern when working with a State pattern. For a bit of wisdom on the subject, here's an excerpt from the GoF book on how to think about the trade-off of either approach:

"Who controls the state transitions?" The State pattern doesn't specify which participant [either Subject or State] defines the criteria for state transitions. If the criteria are fixed, then they can be implemented entirely in the [Subject]. It is generally more flexible and appropriate, however, to let the [State] subclasses themselves specify their successor state and when to make the transitions."

QUIZ QUESTIONS

You should consider using a State pattern if your class

1. Might change significantly upon a condition being met
2. Isn't well-suited to multi-level inheritance
3. Has a lot of conditional statement
4. None of the above

Which class should be responsible for changing State?

1. State
2. Subject
3. Both
4. Either

Which of the following examples is a good candidate for a State pattern?

1. A text editor that allows you to change fonts and styles
2. A video game character that can change into a variety of movement and attack methods
3. A website that should remember user input from a user's previous page request
4. Both A and C
5. Both B and C

Which of the following are potentially eliminated by a State object?

1. A large Subject class
2. Dependencies between a Subject and its State
3. Conditionals that determine behavior
4. Both A and C
5. Both B and C
6. None of the above

Which C# language feature is inherently well-suited to representing states?

1. Events
2. Enums
3. Strings
4. All of the above

QUIZ ANSWERS

1. **(1) Might change significantly upon a condition being met**
2. **(4) Either**
3. **(2) A video game character that can change into a variety of movement and attack methods**
4. **(4) Both A and C**
5. **(2) Enums**

16. Strategy Pattern

PATTERN TYPE: Behavioral

FREQUENCY OF USE: Medium

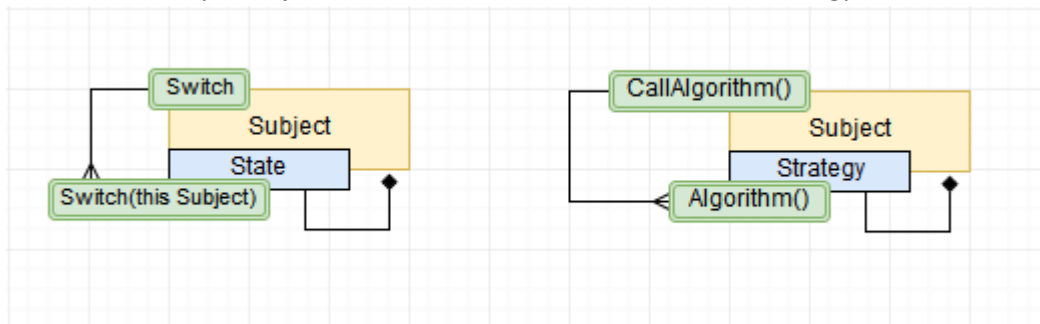
MEMORY HELPER: Set class methods via composition object.

PATTERN OBJECTIVE: To delegate a similar group of method behaviors to outside objects.

STRATEGY PATTERN

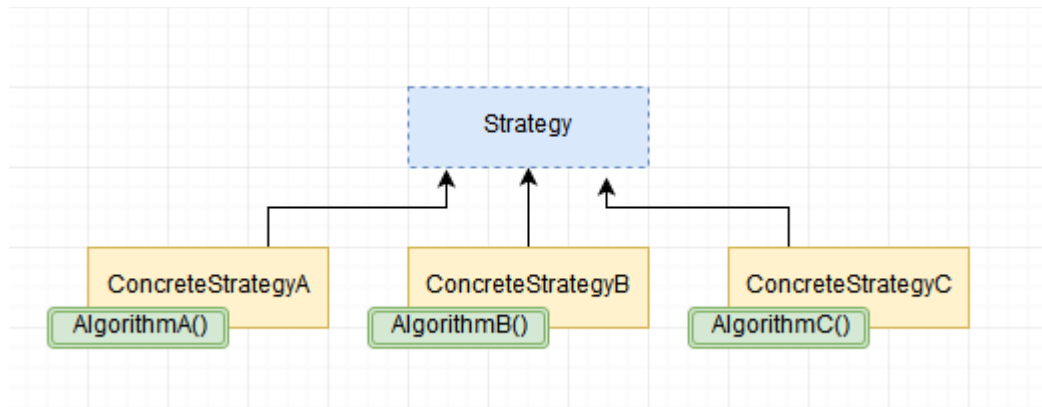
The Strategy pattern is very similar in appearance to the State pattern, but even easier to create and understand. Like the State pattern, you have a Subject which contains a composition reference to an object that it delegates the task of changing its internal behavior. With the State pattern this object is called the “State object” whereas in a Strategy pattern the object is called a “Strategy object.”

Below you will see that the simplest visual representation of these relationships looks almost the same between the way a Subject handles a State vs how it handles a Strategy:



Where the patterns mainly differ is in the fact that a State can take in a Subject and modify it, as well as switch the Subject’s State object to change its behavior. A Strategy pattern, by contrast, only relies on the Strategy object for providing different behaviors. Basically, a State object is much more dynamic than a Strategy object. You can almost think of a Strategy object as like a static class whose main purpose is to hold a set of helper methods.

The methods of a Strategy class often represent different ways of accomplishing a similar group of tasks. The best way to implement these different methods is to define them in separate concrete implementations of an abstract base class, visualized as follows:



The Subject can choose which strategy - and, hence, the type of algorithm/behavior - by instantiating its Strategy object as any one of the available ConcreteStrategies.

STRATEGY SCAFFOLD

We'll start our scaffold version of the Strategy pattern with a simple interface for the Strategy as well as the concrete implementations:

```
interface IStrategy
{
    void Algorithm();
}

class ConcreteStrategyA : IStrategy
{
    public void Algorithm()
    {
        Console.WriteLine("AlgorithmA");
    }
}

class ConcreteStrategyB : IStrategy
{
    public void Algorithm()
    {
        Console.WriteLine("AlgorithmB");
    }
}

class ConcreteStrategyC : IStrategy
{
    public void Algorithm()
    {
        Console.WriteLine("AlgorithmC");
    }
}
```

Next, we will define the Subject with the primary role of delegating which strategy it wants to implement via its Strategy object:

```
class Subject
{
    private IStrategy strategy;
    public Subject(IStrategy strategy)
    {
        this.strategy = strategy;
    }

    public void CallAlgorithm()
    {
        strategy.Algorithm();
    }
}
```

In the client, we wire it up and set the strategy we want to use via the Subject:

```
class Program
{
    static void Main()
    {
        Subject subject = new Subject(new ConcreteStrategyB());
        subject.CallAlgorithm(); //AlgorithmB
    }
}
```

SET EMPLOYEE SORT STRATEGY REAL WORLD EXAMPLE

To build our real-world example of the Strategy pattern, let's suppose you are tasked with creating a company's internal list of employee records that can be returned as a sorted list. We'll delegate the particular manner of sorting to any one of several concrete strategies. Let's begin by defining a basic class that stores an employee record, including fields for an auto-incrementing ID, Name and Salary:

```
class Employee
{
    public Employee(string Name, double Salary)
    {
        incrementID++;
        this.ID = incrementID;
        this.Name = Name;
        this.Salary = Salary;
    }

    private static int incrementID;
    public int ID;
    public string Name;
    public double Salary;
}
```

Next, we'll define the main Subject class, which will maintain a List<Employee> as well as the Strategy object:

```
/* Subject */
class EmployeeList
{
    private List<Employee> list;
    private SortStrategy sortstrategy;

    public void SetSortStrategy(SortStrategy sortstrategy)
    {
        this.sortstrategy = sortstrategy;
    }

    public void Add(Employee employee)
    {
        if (list == null)
            list = new List<Employee>();
        list.Add(employee);
    }

    public void Sort()
    {
        sortstrategy.Sort(list);
    }
}
```

Now we'll define our strategies, starting with an abstract class with the default implementation of iterating over the sorted List<Employee>:

```
/* Strategies */
abstract class SortStrategy
{
    public virtual void Sort(List<Employee> list)
    {
        if (list != null)
            foreach (Employee emp in list)
                Console.WriteLine("ID: {0} - Name: {1} - Salary: {2}",
                    emp.ID,
                    emp.Name,
                    emp.Salary
                );
    }
}
```

For the concrete Strategy implementations we'll just use the available List<> ordering methods. Note that these methods are available through a using System.Linq declaration. Also, the arguments passed into these methods are anonymous method definitions via Lambda Expressions:

```

class SortById : SortStrategy
{
    public override void Sort(List<Employee> list)
    {
        list = list.OrderByDescending(emp => emp.ID).ToList();
        base.Sort(list);
    }
}

class SortByName : SortStrategy
{
    public override void Sort(List<Employee> list)
    {
        list = list.OrderBy(emp => emp.Name).ToList();
        base.Sort(list);
    }
}

class SortBySalary : SortStrategy
{
    public override void Sort(List<Employee> list)
    {
        list = list.OrderByDescending(emp => emp.Salary).ToList();
        base.Sort(list);
    }
}

```

In the client, we can create our set of Employee records and add them to the EmployeeList Subject, set the sorting strategy via the Subject's SetSortStrategy() method and print the result of that chosen strategy:

```

class Program
{
    static void Main()
    {
        // Two contexts following different strategies
        EmployeeList employees = new EmployeeList();
        employees.Add(new Employee("Bill", 55000));
        employees.Add(new Employee("Angela", 42000));
        employees.Add(new Employee("Steve", 39000));
        employees.Add(new Employee("Jim", 48000));

        employees.SetSortStrategy(new SortBySalary());
        employees.Sort();
    }
}

```

QUIZ QUESTIONS

A Strategy appears at first glance as very similar to a:

1. Decorator
2. Factory Method
3. Facade
4. State

The Strategy pattern can be considered a good fit if:

1. You have most invariant behavior with slight differences
2. You need a composite object to manage its behavior
3. You need to represent a variety of differing behaviors
4. None of the above

Which of the following is a good candidate for refactoring into a Strategy pattern?

1. A large class that has too many methods within it
2. An object in a game that changes depending on conditions being met
3. A method with multiple conditionals that determine how to process data
4. None of the above

Which of the following is true about the Subject in a Strategy pattern?

1. It doesn't need to know the Strategy being implemented
2. It has direct knowledge of its Strategy
3. It lets the Strategy object determine itself
4. It dictates how the strategy is implemented

How does the Strategy pattern provides an alternative to inheritance?

1. It doesn't
2. It creates a lot of objects
3. Behavior becomes abstracted away
4. Behavior becomes "outsourced" to an outside object

Which of the following is a potential problem with using the Strategy pattern?

1. Clients can become coupled to a large amount of Strategies
2. Strategies must implement a uniform interface even if they don't require all of its parameters
3. It can trade one potentially monolithic class (Subject) for another (Strategy)
4. As you add concrete Strategies it can grow in complexity
5. A and B are both true
6. B and C are both true
7. D and A are both true

QUIZ ANSWERS

1. **(4) State**
2. **(3) You need to represent a variety of differing behaviors**
3. **(3) A method with multiple conditionals that determine how to process data**
4. **(1) It doesn't need to know the Strategy being implemented**
5. **(4) Behavior becomes "outsourced" to an outside object**
6. **(5) A and B are both true**

17. Observer Pattern

PATTERN TYPE: Behavioral

FREQUENCY OF USE: High

MEMORY HELPER: Subject notifies observers (events)

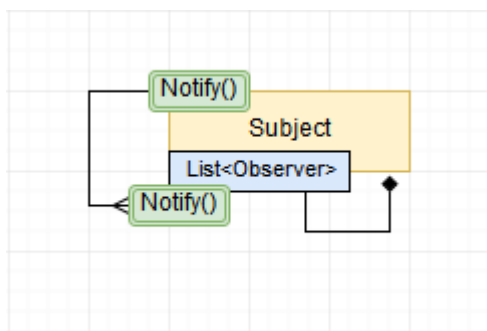
PATTERN OBJECTIVE: A system in which an object can alert its dependencies upon a condition being met. The dependencies can carry out some function in response to the notification.

OBSERVER IN C#

If you've been programming with C# for any length of time then the Observer pattern is probably something you're already familiar with. That's because the C# language has the Observer pattern built-in via **events**. The typical nomenclature for this pattern is to have a single class called the “**Subject**” which needs to notify a series of classes called “**Observers**” upon the firing of some special event. The notification gets sent to the Observers by calling their method that has been “subscribed” to receive notification of the Subject's event.

PRIMITIVE VERSION USING LIST<OBSERVER>

When the GoF described this pattern they did so without using events but rather a more primitive concepts available to most languages. This more universal formulation had a **Subject** maintain a **List<Observer>** that all need to be notified when a certain condition is met. Notifying the Observers becomes a simple matter of the Subject's **Notify()** method *iterating through the List<Observer>* and calling each Observer's own **Notify()** method. You can visualize the idea as follows:



Below is a scaffold representation of this primitive version of the pattern:

```
class Program
{
    static void Main()
    {
        Subject subject = new Subject("The Subject");
        AbstractObserver observerA = new ObserverA();
        AbstractObserver observerB = new ObserverB();

        subject.AddObserver(observerA);
        subject.AddObserver(observerB);
        subject.Notify();
    }
}

/* SUBJECT */
class Subject
{
    private List<AbstractObserver> observers;
    public string Name;

    public Subject (string Name)
    {
        this.Name = Name;
        observers = new List<AbstractObserver>();
    }

    public void Notify()
    {
        if (observers != null)
            foreach (AbstractObserver o in observers)
                o.Notify(this);
    }

    public void AddObserver(AbstractObserver observer)
    {
        observers.Add(observer);
    }

    public void RemoveObserver(AbstractObserver observer)
    {
        observers.Remove(observer);
    }
}
```

```

/* OBSERVERS */
abstract class AbstractObserver
{
    public virtual void Notify(Subject sender)
    {
        Console.WriteLine(this.GetType().Name +
            " received notification from " + sender.Name
        );
    }
}

class ObserverA : AbstractObserver { }
class ObserverB : AbstractObserver { }

```

You may have noticed that the Subject's Notify() method passes in a reference to itself to the Observer's Notify() method in order for the Observer to know about the Subject.

```

public void Notify()
{
    if (observers != null)
        foreach (AbstractObserver o in observers)
            o.Notify(this);
}

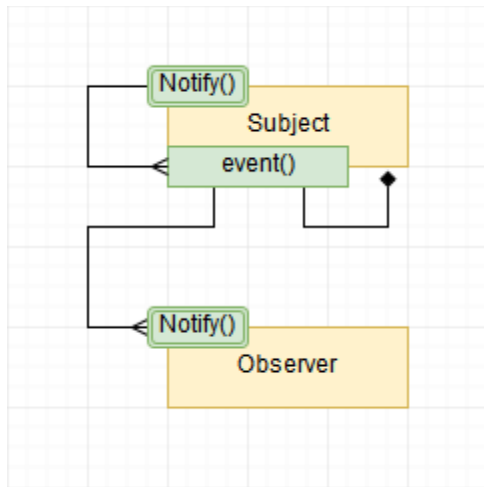
```

This is optional, but worth keeping this mind when we move on to the default event system in C#.

BUILT-IN VERSION USING C# EVENTS

When using C#'s event system to handle Subject/Observer behavior, there isn't a List<> reference to the Observer objects held in the Subject but rather a "multicast delegate" reference to all of the Observers' Notify() methods. This is housed in the Subject's event which matches the signature of the Observers' methods. And instead of the Subject's Notify() method iterating over the Observers Notify() methods, we see the Subject's Notify() "raise" the event which in turn calls all of the Observers Notify() methods that are subscribed to it.

Compare using built-in events to the primitive version:



Here's a scaffold representation using C# events:

```
class Program
{
    static void Main()
    {
        Subject subject = new Subject("The Subject");
        AbstractObserver observerA = new ObserverA();
        AbstractObserver observerB = new ObserverB();

        subject.Handler += observerA.Notify;
        subject.Handler += observerB.Notify;
        subject.Notify();
    }
}

/* SUBJECT */
class Subject
{
    public string Name;
    public event EventHandler Handler;
    public Subject(string Name)
    {
        this.Name = Name;
    }
    public void Notify()
    {
        if (Handler != null)
            Handler(this, EventArgs.Empty);
    }
}
```

```

/* OBSERVERS */
abstract class AbstractObserver
{
    public virtual void Notify(object sender, EventArgs e)
    {
        Subject s = sender as Subject;
        Console.WriteLine(this.GetType().Name +
            " received notification from " + s.Name
        );
    }
}

class ObserverA : AbstractObserver { }
class ObserverB : AbstractObserver { }

```

DEFAULT EVENTHANDLER

You don't have to use C#'s default **EventHandler** type but rather can define your own method signature by creating a delegate and using that as the event's Type, EG:

```
public event MyDelegate Handler;
```

Also, you may have noticed that we 'got away' with using the default EventHandler by casting the base object to a Subject within the Observers' Notify() method:

```

public virtual void Notify(object sender, EventArgs e)
{
    Subject s = sender as Subject;
    Console.WriteLine(this.GetType().Name +
        " received notification from " + s.Name
    );
}

```

You may have also noticed the optional **EventArgs** parameter (optional since we can avoid it by passing in **EventArgs.Empty**). This class is mostly a placeholder for you to inherit and create your own special set of event arguments you might want to pass in. To make use of it you can just inherit from it and send the information through the event as follows:

```

class MyEventArgs : EventArgs
{
    public DateTime TimeStamp = DateTime.Now;
}

```

```

/* SUBJECT */
class Subject
{
    public string Name;
    public event EventHandler Handler;
    public Subject(string Name)
    {
        this.Name = Name;
    }
    public void Notify()
    {
        if (Handler != null)
            Handler(this, new MyEventArgs());
    }
}

/* OBSERVERS */
abstract class AbstractObserver
{
    public virtual void Notify(object sender, EventArgs e)
    {
        Subject s = sender as Subject;
        MyEventArgs args = e as MyEventArgs;
        Console.WriteLine(this.GetType().Name +
            " received notification from " + s.Name + " at time: " + args.TimeStamp
        );
    }
}

```

REAL WORLD OBSERVER EXAMPLE

For our real-world example we'll be creating a product management system where the product is a Subject that needs to notify its Observers when its demand has changed. For example, if the product become low in demand it will notify a PriceAdjustment Observer (so that it can adjust the price accordingly) as well as notify an EmailSupplier Observer (which would send out an email to presumably halt further shipments).

We'll start by defining our own custom delegate signature that we'll be using for our event, passing in a subject rather than just a base object:

```

delegate void StateChangeHandler(ISubject subject);

```

And we'll also define a simple enum to keep track of the demand status for the product:

```

enum DemandStatus
{
    Normal, High, Low
}

```

Next we'll define a simple interface for the product that requires a product to have a StateChangeHandler event, a DemandStatus, Price and Name:

```
interface ISubject
{
    event StateChangeHandler OnStateChange;
    DemandStatus State { get; set; }
    float Price { get; set; }
    string Name { get; set; }
}
```

We could, of course, define multiple products but for simplicity we'll just create a single base product. We'll implement the event as a property, which uses the add/remove syntax rather than the typical get/set. Meanwhile, in the setter for the DemandStatus we'll trigger the event and notify the observers only if the status has changed:


```

class Product : ISubject
{
    public string Name { get; set; }
    public float Price { get; set; }
    public Product(string Name, float Price)
    {
        this.Name = Name;
        this.Price = Price;
    }

    private event StateChangeHandler _onStateChange;
    public event StateChangeHandler OnStateChange
    {
        add { _onStateChange += value; }
        remove { _onStateChange -= value; }
    }

    private DemandStatus _status;
    public DemandStatus Status
    {
        get { return _status; }
        set
        {
            if (_status == value)
                return;
            else
            {
                _status = value;
                notify(); //Inform Observers of this Subject's status change.
            }
        }
    }

    private void notify()
    {
        if (_onStateChange != null)
            _onStateChange(this);
    }
}

```

Next we'll turn to defining our Observers, starting with the interface and abstract base class with some basic implementation details for registering the Observer's Notify method to the Subject's event:

```

/* OBSERVERS */
interface IObserver
{
    void Register(ISubject subject);
    void Notify(ISubject subject);
}

```

```

abstract class Observer : IObserver
{
    public virtual void Register(ISubject subject)
    {
        subject.OnStateChange += Notify;
    }
    public virtual void Notify(ISubject subject)
    {
        Console.WriteLine("{0} noticed {1}'s {2} changed to {3}",
            this.GetType().Name,
            subject.Name,
            subject.State.GetType().Name,
            subject.State
        );
    }
}

```

And finally, we'll define our concrete Observers for emailing the supplier and adjusting the price upon being notified of the demand status change:

```

class EmailSupplier : Observer
{
    public override void Notify(ISubject subject)
    {
        if (subject.State == DemandStatus.Low)
        {
            base.Notify(subject);
            Console.WriteLine("...Emailing Supplier of status" + "\n");
        }
    }
}

```

```

class PriceAdjustment : Observer
{
    public override void Notify(ISubject subject)
    {
        base.Notify(subject);
        if (subject.State == DemandStatus.High)
        {
            subject.Price *= 1.2f;
            Console.WriteLine("...Price adjusted +25% to " + subject.Price + "\n");
        }
        else if (subject.State == DemandStatus.Low)
        {
            subject.Price /= 1.2f;
            Console.WriteLine("...Price adjusted +25% t " + subject.Price + "\n");
        }
        else
        {
            Console.WriteLine("...No price adjustment \n");
        }
    }
}

```

We can finalize wiring up the Subject with its Observers in a client class:

```

class Program
{
    static void Main()
    {
        Subject subject = new Subject("The Subject");
        AbstractObserver observerA = new ObserverA();
        AbstractObserver observerB = new ObserverB();

        subject.Handler += observerA.Notify;
        subject.Handler += observerB.Notify;
        subject.Notify();
    }
}

```

QUIZ QUESTIONS

An “Observer” might be a misnomer because:

1. It doesn't really know anything about the Subject
2. It isn't just passively Observing but “acting” as well
3. It's not constantly monitoring the Subject
4. None of the above

What's the main difference between the GoF Observer pattern and the C# Event's based version?

1. Events solve the problem better
2. Events carry less dependencies since they hold methods rather than objects
3. Events have built in Observer behavior
4. All of the above

Which of the following equally describe the relationship between a Subject/Observer

1. Event/Listener
2. Publisher/Subscriber
3. Reporter/Receiver
4. All of the above

An Event's delegate signature supports loose coupling because:

1. An EventHandler isn't an object dependency
2. It's just an interface for a single method signature
3. It actually doesn't
4. None of the above

QUIZ ANSWERS

1. **(2) It isn't just passively Observing but "acting" as well**
2. **(3) Events have built in Observer behavior**
3. **(4) All of the above**
4. **(2) It's just an interface for a single method signature**

18. Command Pattern

PATTERN TYPE: Behavioral

FREQUENCY OF USE: High

MEMORY HELPER: Encapsulate actions in an object

PATTERN OBJECTIVE: To create abstract representations of command-like behaviors (Command class) that can be separated from classes that act upon those commands (Receiver class). This can provide a layer of abstraction in a system that doesn't know precisely what kind of commands it will have to process. Also, by parceling commands into objects, they can be cataloged in interesting ways such as logging a history of actions or allowing undo/redo procedures.

PREAMBLE: The Command pattern is potentially one of the most flexible and difficult to grasp patterns you will come across. This is partly due to the wide array of problem domains it aims to solve as well as its variety of possible implementations. The nucleus of the pattern, in any form it might take, centers around the encapsulation of "command-like" behavior within an object.

We've certainly seen a host of patterns which outsource behavior to outside objects so, at first glance, a scaffold Command pattern might appear to be a trivial - or even pointless - variation of the same theme. But by focusing on what makes an object "command-like" its novelty should become evident.

COMMAND SCAFFOLD

We'll begin by defining the Receiver interface along with the concrete Receiver class, which will end up receiving commands from a Command instance:

```
//Receivers
interface IReceiver
{
    void ActionUponCommand();
}

class Receiver : IReceiver
{
    public void ActionUponCommand()
    {
        Console.WriteLine(this.GetType() + " ActionUponCommand() called!");
    }
}
```

Next, we'll turn to defining the Command interface and ConcreteCommand class which takes in a Receiver and calls it's "action" method in its ExecuteCommand() method:

```
//Command
interface ICommand
{
    void ExecuteCommand();
}
```

```

class ConcreteCommand : ICommand
{
    Receiver receiver;
    public ConcreteCommand(Receiver receiver)
    {
        this.receiver = receiver;
    }
    public void ExecuteCommand()
    {
        receiver.ActionUponCommand();
    }
}

```

And we'll also define an Invoker class which takes a Command and calls its ExecuteCommand() method. Optionally, you can store the Command (or List<Command>) in the Invoker to perform bookkeeping actions later, as we'll see in the real-world code in a moment:

```

class Invoker
{
    private ICommand command;
    public void SetCommand(ICommand command)
    {
        this.command = command;
    }
    public void InitiateCommand()
    {
        command.ExecuteCommand();
    }
}

```

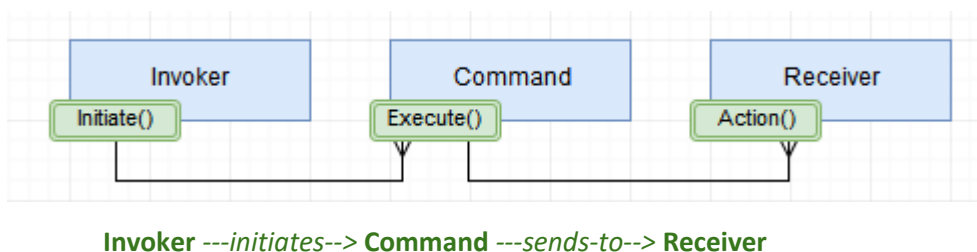
The entire process can be kicked off in a client class by instantiating the Invoker, Command and Receiver as follows:

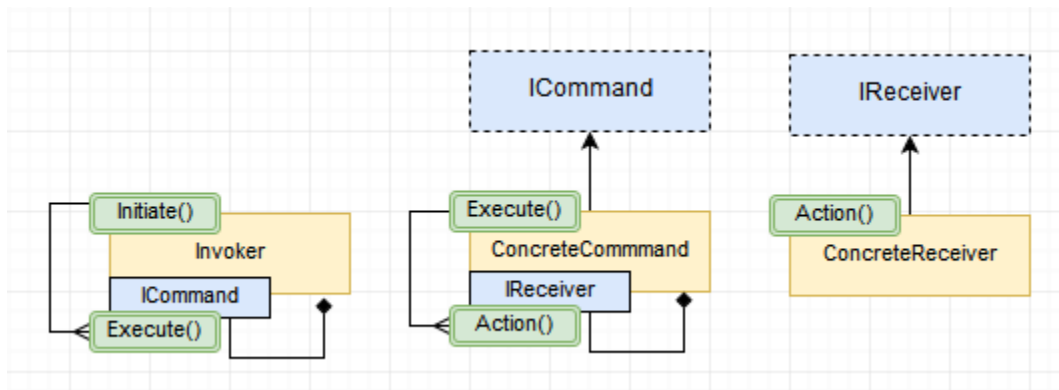
```

class Program
{
    static void Main()
    {
        Invoker invoker = new Invoker();
        invoker.SetCommand(new ConcreteCommand(new Receiver()));
        invoker.InitiateCommand();
    }
}

```

You can visualize how the three key parts relate to each other as follows:





MILITARY HIERARCHY ANALOGY

It might be helpful to picture a military command hierarchy when trying to memorize, or recall, the basic relationships between the three main classes of the pattern:

- **Invoker (Chief)** - Sends raw information to the Command (General)
- **Command (General)** - Takes info from the Invoker (Chief) and passes it to the Receiver (Cadet)
- **Receiver (Cadet)** - Receives a command from the Command (General), and acts upon it

What stands out in this sequence is how the Command can be by-passed and still result in the same basic kind of behavior. After all, an Invoker could send its information straight to the Receiver, so why bother with a Command middle-man? If we by-passed the Command class in this way we wouldn't be able to parcel the commands into objects, store them in the Invoker for possible undo/redo procedures or other kinds of bookkeeping. Also, you may not be able to easily substitute new and unforeseen Commands at a later date without heavily modifying the Invoker or creating a plethora of subclasses to handle different Command types.

REAL-WORLD COMMAND EXAMPLE (CALCULATOR)

The scaffold representation we looked at provides a very rough overview of how the three main classes operate, but struggles to instill an appreciation for the benefits of keeping them separate. Our real-world example of creating a calculator application that takes in commands should go much further in demonstrating a few of the main benefits. Designing a calculator is one of the most intuitive ways of demonstrating the Command pattern. Here are the roles of the three main classes:

- **Invoker (User)** - Imports user input and stores into a `List<Command>`, initiates Command
- **Command (Operation)** - Stores user input as Command, executes normal/opposite to Receiver
- **Receiver (Calculator)** - Receives user input as Command, performs calculation based on it

We're going to need to represent the four basic types of arithmetic operations that our calculator will perform upon command. Since these operations are constants, it'll be easier to refer to them as members of an enum:


```
enum Operation
{
    Add,
    Subtract,
    Multiply,
    Divide
}
```

Next, we'll define our interface that abstractly represents key parts of the Command. The following defines a **double n** along with a specific **Operation** type. For instance a Command might hold the values

```
n = 4;
operation = Operation.Add;
```

The Receiver will later be able to parse this information (quite intuitively) and **add 4** to its existing sum. Meanwhile, we also include an **Execute()** method which will pass the information to the Receiver, as well as an **ExecuteOpposite()** method that we will later use to implement undo procedures:

```
interface ICommand
{
    double n { get; set; }
    Operation operation { get; set; }

    void Execute();
    void ExecuteOpposite();
}
```

Here is how the concrete Command class implements these features:

```
class Command : ICommand
{
    public double n { get; set; }
    public Operation operation { get; set; }
    IReceiver receiver;

    public Command(Operation operation, double n, IReceiver receiver)
    {
        this.n = n;
        this.operation = operation;
        this.receiver = receiver;
    }

    public void Execute()
    {
        this.receiver.Calc(this);
    }

    public void ExecuteOpposite()
    {
        //TBD
    }
}
```

You'll notice that the Command's Execute() method passes the command onto the Receiver by referencing itself. Let's turn to defining the Receiver and its interface in order to make this possible.

```
interface IReceiver
{
    void Calc(ICommand command);
}
```

It should be self-explanatory how the Receiver's Calc() method receives the Command and performs the calculations based upon it:

```
class Receiver : IReceiver
{
    private double sum;
    public void Calc(ICommand command)
    {
        switch(command.operation)
        {
            case Operation.Add:
                sum += command.n;
                break;
            case Operation.Subtract:
                sum -= command.n;
                break;
            case Operation.Multiply:
                sum *= command.n;
                break;
            case Operation.Divide:
                sum /= command.n;
                break;
            default:
                sum = 0;
                break;
        }

        Console.WriteLine(sum);
    }
}
```

Turning now to the Invoker, we'll have it store a Receiver instance (the calculator) as well as List<ICommand> for storing undo and redo operations. The Do() method takes in user input, sends it to the Command along with the Receiver reference, whereby the Command then takes the responsibility of passing it to the Receiver via that reference. We then add the Command to the undoList and initialize the redoList to overwrite any prior redo procedures that may have been queued.

```

class Invoker
{
    IReceiver calculator = new Receiver();
    List<ICommand> undoList = new List<ICommand>();
    List<ICommand> redoList = new List<ICommand>();
    public void Do(Operation operation, double n)
    {
        Command command = new Command(operation, n, calculator);
        command.Execute();
        undoList.Add(command);
        redoList = new List<ICommand>();
    }
}

```

Turning now to implementing undo procedures, we first write a simple conditional that escapes the entire Undo() method if there are no undo procedures enqueued in undoList. Otherwise, we'll grab the last Command entered in the List<>, execute the opposite of that Command and then pop off that procedure from the undoList and move it to the redoList instead:

```

class Invoker
{
    ...
    public void Undo()
    {
        if (undoList.Count == 0)
        {
            Console.WriteLine("NO OPERATION TO UNDO");
            return;
        }

        ICommand lastCommand = undoList[undoList.Count - 1];
        Console.WriteLine("UNDO back to ");

        lastCommand.ExecuteOpposite();
        redoList.Add(lastCommand);
        undoList.Remove(lastCommand);
    }
}

```

At this point it's necessary to define how the Command knows to execute its polar opposite, so we'll return to the concrete Command class to define the ExecuteOpposite() method. We accomplish this by substituting the opposite arithmetic operation than the one that the Command originally contained, rebundling it into a new Command and sending that Command to the Receiver:

```

class Command : ICommand
{
    ...
    public void ExecuteOpposite()
    {
        Operation opposite;
        switch (this.operation)
        {
            case Operation.Add:
                opposite = Operation.Subtract;
                break;
            case Operation.Subtract:
                opposite = Operation.Add;
                break;
            case Operation.Multiply:
                opposite = Operation.Divide;
                break;
            case Operation.Divide:
                opposite = Operation.Multiply;
                break;
            default:
                opposite = Operation.Add;
                break;
        }
        this.receiver.Calc(new Command(opposite, this.n, this.receiver));
    }
}

```

Turning back one more time to the Invoker we'll need to define the Redo() operation, which will be similar to the Undo() operation, except it pulls the last Command from the redoList and pops it back onto the undoList at the end of the process. It also, of course, executes the Command normally using Execute():

```

class Invoker
{
    ...
    public void Redo()
    {
        if (redoList.Count == 0)
        {
            Console.WriteLine("NO OPERATION TO REDO");
            return;
        }

        ICommand lastCommand = redoList[redoList.Count - 1];
        Console.Write("REDO back to ");
        lastCommand.Execute();
        undoList.Add(lastCommand);
        redoList.Remove(lastCommand);
    }
}

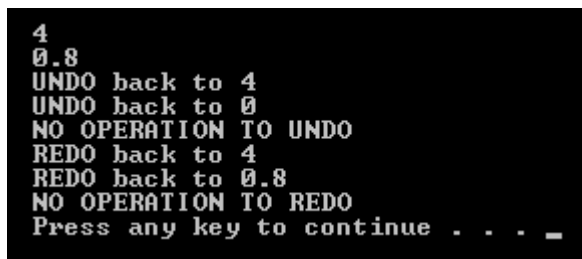
```

In a client, we can create a user (which creates the calculator as well), send it calculation Commands as well as perform Undo/Redo procedures as follows:

```
class Program
{
    static void Main()
    {
        Invoker user = new Invoker();
        user.Do(Operation.Add, 4);
        user.Do(Operation.Divide, 5);

        user.Undo();
        user.Undo();
        user.Undo();

        user.Redo();
        user.Redo();
        user.Redo();
    }
}
```



```
4
0.8
UNDO back to 4
UNDO back to 0
NO OPERATION TO UNDO
REDO back to 4
REDO back to 0.8
NO OPERATION TO REDO
Press any key to continue . . . _
```

FURTHER CONSIDERATIONS

The implementation we covered here is one of many possible use cases for this pattern. The approach you take when using the pattern depends heavily on what the application demands. The following are some important considerations and scenarios that might alter how you apply the pattern.

Irreversible Operations: We looked at how we can reverse commands to perform opposite calculations when using an undo procedure. This is quite simple with arithmetic operations but not so easy in other cases, and impossible in others. For example, when implementing undo operations for commands in a photo editing app, how would you reverse irreversible procedures such as applying blur on an image or reducing the resolution? The command wouldn't be able to provide this logic.

In such cases, you would probably want to save the "state" of the image as a historical snapshot rather than the Command operations themselves. This would, of course, increase memory usage since each undo level results in another copy of the image in memory. If we implemented this scenario for our calculator app, we would have simply made the `undoList` and `redoList` a `List<int>` instead of `List<Command>`, the `int` representing the sum "state" that we would pull from the Receiver and store as the result of the Command. This creates the same end result, but with a very different implementation.

Command Complexity: Another issue might be if we wanted to add more complex calculations, like the ones you'd find in a scientific calculator (log, sqr root, etc). In this case, it might be better to house the calculation logic in Command *subclasses* rather than in the Receiver, leaving the Receiver to only handle displaying the sum instead. Doing so would keep the classes small, handling a single task rather than having one class to handle all possible calculations in a giant switch() conditional.

Object-Oriented Callback Methods: A use case that might not appear obvious is storing behaviors in objects that can be called upon some kind of event. This is actually built into C#, as you can easily store callback functions as delegate methods and tie them into any kind of event - like a menu or button click, for instance. In other words, entire procedures can be treated as first-class objects. In this way, the Command would be the delegate itself, executing whatever method it holds. Alternatively, you can store a Command object (just like the scenario we looked at), which might include state information as well as multiple method definitions if a delegate callback has a difficult time solving the problem at hand.

Delaying Requests: Another possible application is delaying requests into a queue in order to be executed later. Because Commands are objects that can be stored in a List<>, this gives you the developer the power to determine how and when a request is performed in cases where it has to be delayed. One possible application for this is buffering user input in a video game. Normally, user input is accepted on a frame and executed in the next frame. However, in some cases you would want to store that user input command and execute it later. One such example is a game that has attack "combos" where a user might input an attack command *before* the previous one completed. Without buffering the command, the follow-up attack never comes out, but by delaying the early command you can ensure that the attacks string together.

QUIZ QUESTIONS

How could a Command pattern support undo procedures?

1. Storing the entire procedure as a Command object
2. Storing the relevant information in a Command object
3. Storing relevant state information resulting from a Command
4. All of the above
5. None of the above

What are typical roles of an Invoker?

1. Manage Commands, cataloging them and so forth
2. Adding a layer of separation between Commands and Receivers
3. Delay execution of the Command
4. All of the above
5. None of the above

Which class should be responsible for carrying out the Command's resulting implementation?

1. Receiver
2. Command
3. Invoker
4. It varies

What is the order of method execution?

1. Invoker.Initiate(), Receiver.Action(), Command.Execute()
2. Invoker.Initiate(), Command.Execute(), Receiver.Action()
3. Receiver.Action(), Invoker.Initiate(), Command.Execute()
4. It varies

QUIZ ANSWERS

1. **(4) All of the above**
2. **(4) All of the above**
3. **(4) It varies**
4. **(2) Invoker.Initiate(), Command.Execute(), Receiver.Action()**

19. Iterator Pattern

PATTERN TYPE: Behavioral

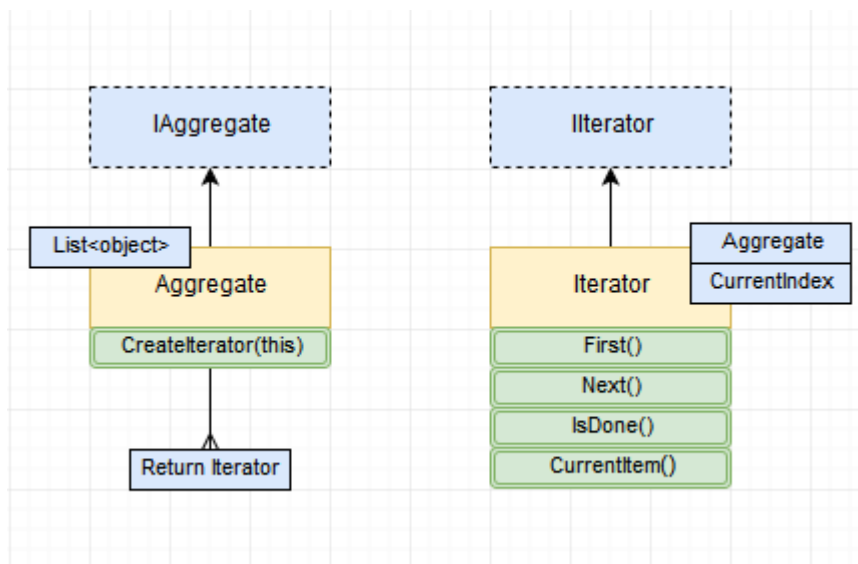
FREQUENCY OF USE: High

MEMORY HELPER: Separate Iteration Procedures from a Collection

PATTERN OBJECTIVE: To allow highly customizable iteration behavior against aggregate collections (Arrays, ArrayLists, Lists, etc) by housing the aggregate data separate from its iteration procedures. The use of interfaces to represent the aggregate data as well as the iteration behaviors allows for flexible and modular approach to iterating over collections.

PREAMBLE: The Iterator pattern is surprisingly common given that most languages have built-in procedures for iterating over collections of data. The C# language, and the .NET Framework in particular, have a variety of ways of storing and iterating over aggregate data (IE: collections) like Arrays, ArrayLists and Lists.

It may seem as though the language has everything you would ever need to iterate over collections of data and in a lot of ways this is true. However, by implementing the Iterator pattern, you potentially gain more control over data collections as well as the iteration procedures you perform on that data. Before we take a closer look, let's examine a scaffold example of the Iterator pattern in C# visualized as follows:



ITERATOR SCAFFOLD

We'll start by creating simple interfaces for the two main concrete Aggregate and Iterator classes:

```
interface IAggregate
{
    IIterator CreateIterator();
}

interface IIterator
{
    object First();
    object Next();
    bool IsDone();
    object CurrentItem();
}
```

The **ConcreteAggregate** will essentially be a class that wraps around an underlying aggregation type. You can choose any underlying type you wish - **Array**, **ArrayList**, **List**, etc - but we'll keep this example as primitive as possible and simply use an array to store the aggregate data as **object[] items**:

```
class ConcreteAggregate : IAggregate
{
    private object[] items;
    public ConcreteAggregate(object[] items)
    {
        this.items = items;
    }

    public object this[int index]
    {
        get { return items[index]; }
        set { items[index] = value; }
    }

    public int Count
    {
        get { return items.Length; }
    }

    public IIterator CreateIterator()
    {
        return new ConcreteIterator(this);
    }
}
```

This notation might stand out as odd:

```
public object this[int index]
{
    get { return items[index]; }
    set { items[index] = value; }
}
```

However, it simply allows us access the underlying **object[] items** index (which is otherwise private) via an indexer postfixed to the enclosing type. In other words:

```
ConcreteAggregate aggregate;  
aggregate[0] = value;  
aggregate[1] = value;  
aggregate[2] = value;
```

Let's now turn to defining the **ConcreteIterator**, which will be instantiated after the **ConcreteAggregate** instant calls its **CreateIterator()** method (which passes the **ConcreteAggregate** into the **ConcreteIterator** constructor and returns that Iterator). This will be clearer in a moment:

```
class ConcreteIterator : IIterator  
{  
    private ConcreteAggregate aggregate;  
    private int current = 0;  
  
    public ConcreteIterator(ConcreteAggregate aggregate)  
    {  
        this.aggregate = aggregate;  
    }  
    //Return first item in aggregate  
    public object First()  
    {  
        return aggregate[0];  
    }  
    //Return next item in aggregate unless it's at the last item  
    public object Next()  
    {  
        return (current < aggregate.Count - 1) ? aggregate[current++] : null;  
    }  
    //Return current item in aggregate  
    public object CurrentItem()  
    {  
        return aggregate[current];  
    }  
    //Returns true when last item in aggregate is reached  
    public bool IsDone()  
    {  
        return current >= aggregate.Count - 1;  
    }  
}
```

Once you have the **ConcreteIterator** instantiated, you shouldn't directly have to manipulate the **ConcreteAggregate** object, instead letting the **ConcreteIterator**'s methods perform all collection-related tasks upon its own private instance of **ConcreteAggregate aggregate**.

Take special note of the **int current** field here. This value is meant to hold the current index in all operations involving the aggregate collection. Having this independent index "reference" will eventually demonstrate an important aspect of this particular pattern.

Let's now bring this all together in a client and begin to see how it all works:

```

class Program
{
    static void Main()
    {
        IAggregate aggregate = new ConcreteAggregate(
            new object[] { "Item1", "Item2", "Item3", "Item4" }
        );

        // Create Iterator via aggregate
        IIterator iterator = aggregate.CreateIterator();
    }
}

```

So, now we have an Iterator that can handle iteration procedures on the underlying aggregate it encloses. However, the basic methods provided can only do so much; namely, moving through each item one index at a time via **Next()** and perhaps returning that item via **CurrentItem()**;

Let's implement a customized foreach-type looping method that allows for even further customized selection criteria, defining it first in the interface and then fleshing it out in the concrete class:

```

interface IIterator
{
    ...

    void ForEach(Action<ConcreteIterator> action);
}

```

Most of this logic simply replicates a default foreach() loop. However, the interesting part is in allowing custom actions to be performed, such as specifying selection criteria, within the loop via the **Action<>** input parameter:

```

class ConcreteIterator : IIterator
{
    ...

    public void ForEach(Action<ConcreteIterator> action)
    {
        while (true)
        {
            action(this);

            if (IsDone())
                return;

            Next();
        }
    }
}

```

Going back to the client, let's perform one of many possible criteria we can flexibly implement:

```
class Program
{
    static void Main()
    {
        ...
        iterator.ForEach(a =>
        {
            if (a.CurrentItem().ToString().EndsWith("3"))
                Console.WriteLine(a.CurrentItem());
        });
    }
}
```

Here we passed into the **ForEach()** method a simple **Lambda Expression** that displays all items in the underlying aggregate collection that end with "3". It's worth bearing in mind that this custom ForEach() doesn't reset at the end of the loop. So if you ran this exact sequence over again, it wouldn't return a match since the current index in the Iterator will stay at the last possible index value. To fix this, you can simply reset that value before the loop is done:

```
public void ForEach(Action<ConcreteIterator> action)
{
    while (true)
    {
        action(this);

        if (IsDone())
        {
            current = 0;
            return;
        }

        Next();
    }
}
```

REAL-WORLD ITERATOR CODE

Every programmer knows what it's like to work with data collections, so demonstrating a verbose use case for iteration might not be all that illuminating. What we're going to do instead is assume we have a background data source - such as a database - that gets pulled into a local aggregate object, and then apply the Iterator pattern to it. However, we're going to recruit .NET's built-in Iterator structure via the interfaces **IEnumerable** and **IEnumerator**.

IENUMERABLE AND IENUMERATOR

These built-in interfaces are basic corollaries of an Iterator pattern's **Aggregate (IEnumerable)** and **Iterator (IEnumerator)**.

Being interfaces, they don't provide direct functionality when inheriting from them but rather have a native enforcement of the Iterator pattern. If nothing else, it means you don't have to define these base interfaces yourself and, as an added advantage, the language can treat the resulting collection natively, allowing you to apply native iteration procedures directly to the Aggregate such as in a `foreach()`:

```
IEnumerable aggregate = new Aggregate();

foreach (object item in aggregate)
    Console.WriteLine(item);
```

We'll start by defining our Aggregate class as an IEnumerable. The only requirement from this interface is to return an IEnumerator via **GetEnumerator()**. We'll use this to return an Iterator while passing into the Iterator the Aggregate collection itself:

```
class Aggregate : IEnumerable
{
    private object[] items;

    public object this[int index]
    {
        get { return items[index]; }
        set { items[index] = value; }
    }

    public Aggregate(object[] items)
    {
        this.items = items;
    }

    public int Count
    {
        get { return items.Length; }
    }

    public IEnumerator GetEnumerator()
    {
        return new Iterator(this);
    }
}
```

Turning now to the Iterator class, as an IEnumerator it will have the following methods and properties we'll have to implement for basic iterative procedures:

- **Current** - Property that holds the current index value in the IEnumerator
- **MoveNext()** - Moves current index to next value unless at end of the collection
- **Reset()** - Resets the current index back to starting position

```

class Iterator : IEnumerator
{
    private Aggregate items;
    public Iterator(Aggregate items)
    {
        this.items = items;
    }

    private int _current = -1;
    public object Current
    {
        get { return items[_current]; }
    }

    public bool MoveNext()
    {
        if (_current < items.Count - 1)
        {
            _current++;
            return true;
        }
        else
            return false;
    }

    public void Reset()
    {
        _current = -1;
    }
}

```

You might have noticed that the default position for the index is at -1, meaning if you were to get the current value when the index is at its default position, it would throw an **IndexOutOfRangeException**. To deal with this you might want to implement custom exception handling (**try/catch**) in the getter, or modify the getter to return the zero index when the current index is at -1. The reason why we don't just make the default position 0 is because we'll iterate over the aggregate using `MoveNext()`, which would skip the zero index unless we start at -1.

POPULATING THE AGGREGATE

Since the underlying structure of the Aggregate is an array of objects, you can populate the aggregate with any kind of object. For our purposes, we'll assume we're pulling data from a database and storing it in a local object. In this case, we'll pull Hockey player statistics that are then added to the aggregate:

```

public class HockeyPlayer
{
    public string Name;
    public int Goals;
    public int Assists;
}

```

We populate the aggregate with **HockeyPlayer** data within a client class and create the iterator to handle this particular collection:

```
class Program
{
    static void Main()
    {
        IEnumerable aggregate = new Aggregate(new HockeyPlayer[3] {
            new HockeyPlayer() { Name = "McDavid", Goals = 30, Assists = 70 },
            new HockeyPlayer() { Name = "Crosby", Goals = 44, Assists = 45 },
            new HockeyPlayer() { Name = "Kane", Goals = 34, Assists = 55 }
        });
        IEnumerator iterator = aggregate.GetEnumerator();
    }
}
```

Using the iterator, we can create an impromptu foreach() type loop using that iterator's methods. As noted earlier, this would be functionally the same as using a native foreach. However, using a foreach would bypass the iterator, as well as any Iterator-specific procedures, so we'll use the Iterator's MoveNext() method as a foreach()-like loop instead:

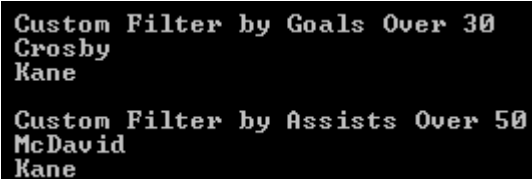
```
static void Main()
{
    ...
    Action CustomForeach = () =>
    {
        Console.WriteLine("Custom Foreach");
        while (iterator.MoveNext())
            Console.WriteLine(((HockeyPlayer)iterator.Current).Name);
        iterator.Reset();
        Console.WriteLine();
    };
    CustomForeach();
}
```

Below is a much more flexible variant that implements dynamic selection criteria on demand. Here we're defining an Action<> method that takes in a string to display a message as well as *another* Action<> method we'll inject later at this method's call:

```
class Program
{
    static void Main()
    {
        ...
        Action<string, Action<HockeyPlayer>> CustomFilter = (msg, act) =>
        {
            Console.WriteLine("Custom Filter by " + msg);
            while (iterator.MoveNext())
                act(iterator.Current as HockeyPlayer);
            iterator.Reset();
            Console.WriteLine();
        };
    }
}
```


We can now call **CustomFilter()** and inject the selection criteria, in this case as an Action that takes in a **HockeyPlayer**, and performs the selection process:

```
class Program
{
    static void Main()
    {
        ...
        CustomFilter("Goals Over 30", hp =>
        {
            if (hp.Goals > 30)
                Console.WriteLine(hp.Name);
        });
        CustomFilter("Assists Over 50", hp =>
        {
            if (hp.Assists > 50)
                Console.WriteLine(hp.Name);
        });
    }
}
```



```
Custom Filter by Goals Over 30
Crosby
Kane

Custom Filter by Assists Over 50
McDavid
Kane
```

IMPORTANT CONSIDERATIONS

The point of using this pattern can be easily missed at multiple junctures. What it comes down to is control and flexibility. To that end, here are some important considerations to keep in mind:

Flexibility via Separation: Evidently, you *could* include the implementation details and the aggregate data in a single Iterator class - binding them together in the same way that a `List<>` and its methods are housed and accessible within the same class. But by portioning the data type (`IEnumerable`, `Aggregate`) as separate from the implementation (iterator, `IEnumerator`) you get a cleaner separation of concerns. Also, and perhaps more importantly, since you don't have to bind the aggregate data type to any particular iteration implementations you can attach different iterators to an aggregate, and different aggregates to an iterator later on. Because they're both decoupled from each other, they become highly flexible and reusable.

Retaining Index Reference: Since each iterator holds an internal index (`Current`), you can run a single aggregate collection against many iterators at the same time. For instance, you can have an aggregate collection where an iterator stops and holds reference to a particular index and then run another iteration with another iterator instance without ever losing that reference. This might be particularly useful if you're dealing with very large collections and need to run through the collection, pause at a particular point (to perform some auxiliary function perhaps), and then resume the iteration.

Efficiency: Keeping index reference can be more efficient than running through the same collection from the beginning each time, effectively retracing indexes that have already been iterated over. This is

analogous to pausing a race and having every racer walk back to the beginning and retrace their steps just to end up at the same point where the race was paused, which is especially inefficient when the travel length is long.

Underlying Collection Type: In the examples we looked at, we chose an array of objects as the underlying collection data type. Obviously you can choose any underlying type you prefer, bearing in mind the built-in tradeoffs and limitations of each type you choose to work with. The point isn't to obscure the underlying data type, but rather to provide your own implementation when iterating over it

QUIZ QUESTIONS

Hiding/encapsulating the underlying collection type has the following consequence:

1. You can't use native iteration procedures like `foreach()`
2. The Iterator doesn't directly know about the underlying type
3. The type could change and the client wouldn't know the difference
4. Both A and C
5. Both B and C

Can an Aggregate have more than one Iterator?

1. Yes
2. No
3. It can but there's no point

What is the .NET equivalent of an Aggregate / Iterator?

1. List / Foreach
2. Object[] / GetEnumerator()
3. IEnumerator / IEnumerable
4. IEnumerable / IEnumerator
5. None of the above

The object that encapsulates an underlying data collections is called an:

1. Iterator
2. Aggregate
3. Array
4. Collection

One of the advantages of using IEnumerable/IEnumerator is:

1. You can use `List<>` as an underlying aggregation type
2. Obviates any need to create your own custom interfaces for the Aggregate/Iterator
3. Provides a single interface that matches existing and inheriting aggregations/collections
4. You inherit most of the functionality you will ever need

QUIZ ANSWERS

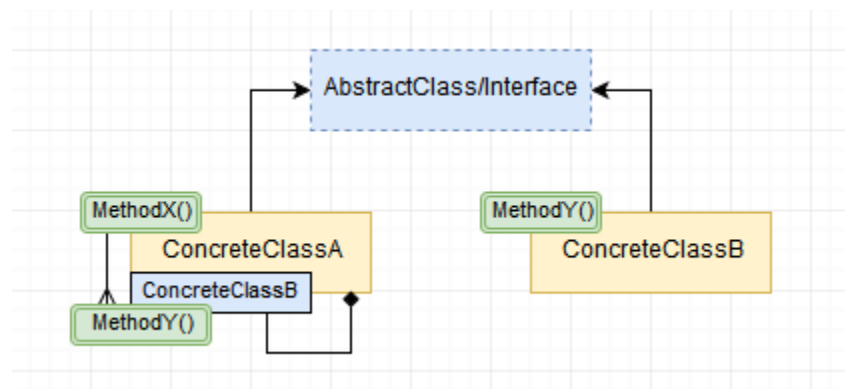
1. **(5) Both B and C**
2. **(1) Yes**
3. **(4) IEnumerable / IEnumerator**
4. **(2) Aggregate**
5. **(3) Provides a single interface that matches existing and inheriting aggregations/collections**

CheatSheet

HOW TO USE THIS CHEATSHEET & DIAGRAMS

The following diagrams aim to depict the important concepts behind each pattern in a simple, succinct way. These diagrams are *not made in UML* (Universal Modeling Language) although they bear some superficial resemblance. If you wish to see UML diagrams of any of these design patterns, they are numerous and easy to find through web search. What you will get here instead is a simplified representation showing **Abstractions vs Concretions**, **Inheritance**, **Object Composition** and **Method Dependencies**, as well as important **Code Flow** usually in the form of how methods call other methods.

The diagrams and their descriptions won't likely be helpful unless you've taken time to understand the corresponding lesson. Once you've understood the pattern, the cheatsheet synopsis should be enough to refresh your memory and communicate the gist of what problem the pattern solves, its typical usage and how its most important mechanics/relations do their work. The diagram below is an example that portrays most of the relevant information you will see in other diagrams.



- **ConcreteClassA** and **ConcreteClassB** inherits from **AbstractClass/Interface**
- **ConcreteClassA** contains an instance of **ConcreteClassB**
- **ConcreteClassA.MethodX()** calls **ConcreteClassB.MethodY()** via the **ConcreteClassB** instance

CREATIONAL vs STRUCTURAL vs BEHAVIORAL

The three main categories in which a pattern falls under communicate the predominant intent behind the pattern. Here is a brief definition of each category's intent:

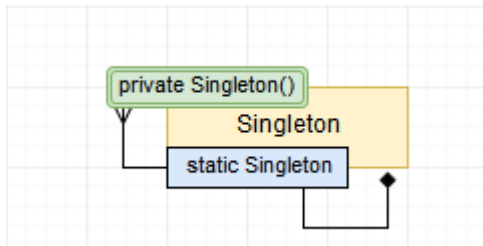
- **Creational** – Handling and abstracting the creation of objects
- **Structural** – Setting up class hierarchies and relationships to enforce a design rule
- **Behavioral** – Creating classes with particular attention to how they handle behavior via methods

SINGLETON (creational)

MEMORY HELPER: Unique Object

PATTERN OBJECTIVE: Limit instantiation of a given class to a single instance, ensuring the uniqueness of the object in the system. Also, providing a global access point to the static instance.

EXAMPLE USAGE: File System object. Video game Player1 object.



BRIEF DESCRIPTION: The Singleton class contains itself via a composition object reference **static Singleton**. A private constructor prevents instantiation, instead relying on access to the static Singleton instance to instantiate itself upon being requested by a client, such as with:

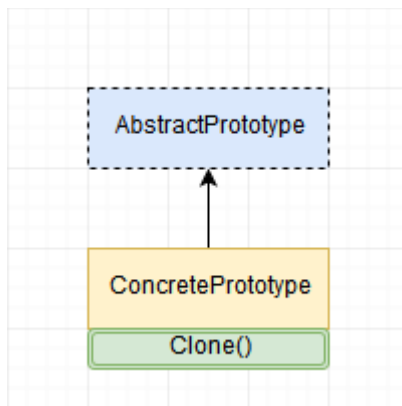
```
Singleton singleton = Singleton.Instance;
```

PROTOTYPE (creational)

MEMORY HELPER: Clone Objects

PATTERN OBJECTIVE: Use a low-cost method of creating new objects by cloning existing objects.

EXAMPLE USAGE: Reducing processing cost when creating many objects, particularly in a short period of time, such as with a bullet spawner in a video game.



BRIEF DESCRIPTION: The `Clone()` method uses `.NET MemberwiseClone()` method to clone the current instance as a shallow copy (which gets copied as a reference, rather than a value type):

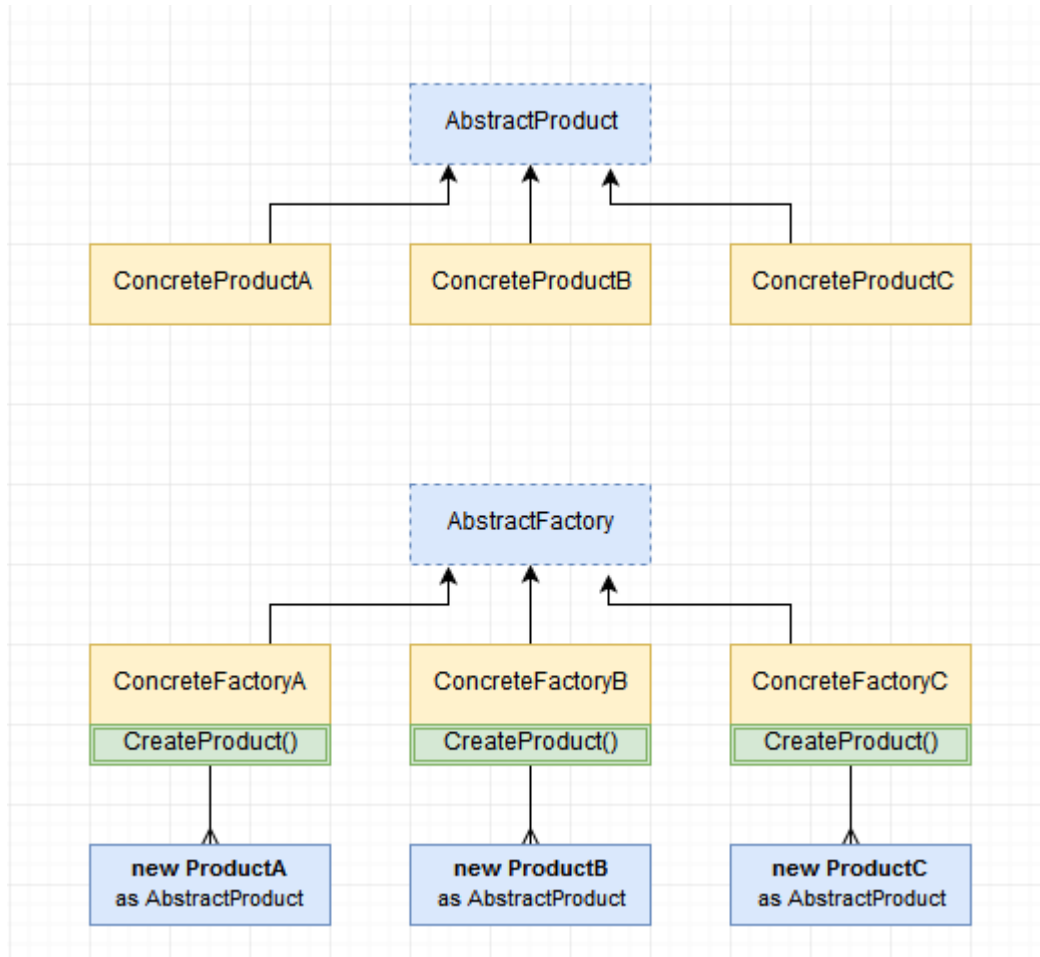
```
AbstractPrototype prototype1 = new ConcretePrototype1();
AbstractPrototype clone1 = prototype1.Clone();
```

FACTORY METHOD (creational)

MEMORY HELPER: Hiding Object Creation

PATTERN OBJECTIVE: Use an interface method for creation of simple objects that share that interface, rather than explicitly using the “new” keyword to create objects wherever clients need them. This keeps object creation details from the rest of the system, facilitating the open/closed principle.

EXAMPLE USAGE: An extendible system of drawing objects, game objects, product objects, etc:



BRIEF DESCRIPTION: The Factories hide object creation of Product objects. A client has no concrete knowledge of the Product:

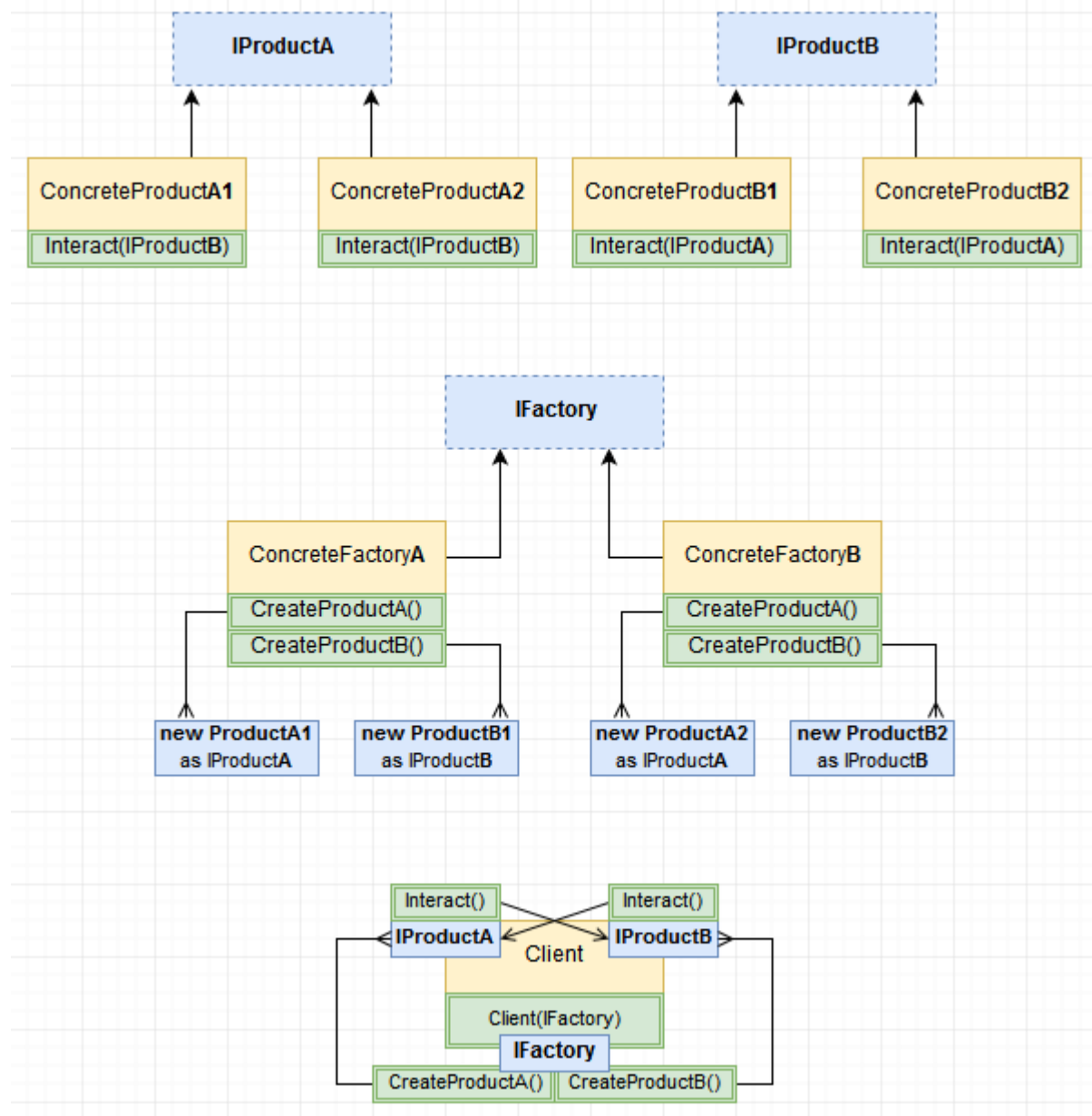
```
IFactory FactoryA = new ConcreteFactoryA();
AbstractProduct ProductA = FactoryA.CreateProduct();
```

ABSTRACT FACTORY (creational)

MEMORY HELPER: Abstracting Creation of Related Objects

PATTERN OBJECTIVE: Use Abstract/Interface definitions or creation of simple objects that have different interfaces and yet need to “interact” with one another, keeping the entire creation/interaction process highly abstracted.

EXAMPLE USAGE: Products that need to “interact” with related Orders in a retail system.



BRIEF DESCRIPTION: Abstract Factories have similar principles of enclosing object creation as a Factory Method, but also create them in “families” of related objects. Each Product object can interact with a family member Product of the opposite interface type. EG: and IProductA can interact with an IProductB. The client, meanwhile, has complete abstract knowledge of the exact objects it has as well as their interactions:

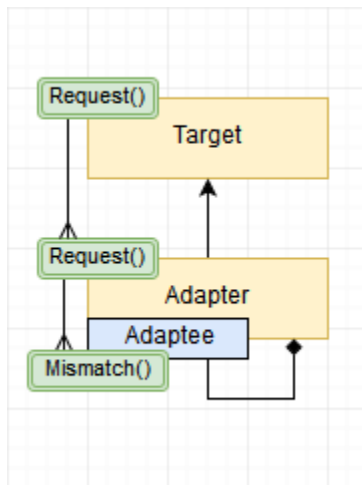
```
class Client
{
    private IProductA productA;
    private IProductB productB;
    public Client(IFactory factory)
    {
        productA = factory.CreateProductA();
        productB = factory.CreateProductB();
    }
    public void DoInteraction()
    {
        productB.Interact(productA);
    }
}
```

ADAPTER (structural)

MEMORY HELPER: Module Converter

PATTERN OBJECTIVE: Concretely retrofit a module within a system that expects a different interface.

EXAMPLE USAGE: Adapting a 3rd party module/library to fit an existing interface (EG: fitting a logging module into an existing system even though the method signatures mismatch)



BRIEF DESCRIPTION: The Adaptee (EG: 3rd party module) has mismatching method signatures, however the intent is to use the Adaptee to replace the Target's existing method calls to similar functionality. The Adapter inherits the Target's method's, matching them, and overrides them to call the underlying Adaptee which the Adapter wraps around:

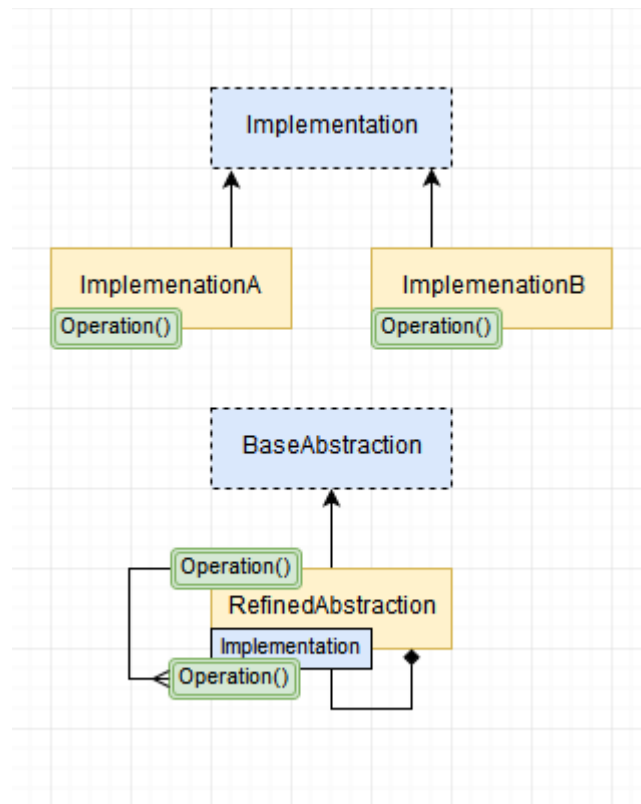
```
Target adapted = new Adapter();
adapted.Request(); //calls Adaptee's Request()
```

BRIDGE (structural)

MEMORY HELPER: Platform Independent Abstraction

PATTERN OBJECTIVE: To abstract classes we are creating in order to separate the abstraction from the implementation, allowing for portability/swapability.

EXAMPLE USAGE: Creating a logging system from the ground-up to allow swapping with different logging modules.



BRIEF DESCRIPTION: In a Bridged system, the Implementation module in the RefinedAbstraction can be swapped with newly derived Implementations. Existing calls to RefinedAbstraction.Operation() will still work because the new Implementation.Operation() fits the same Implementation interface:

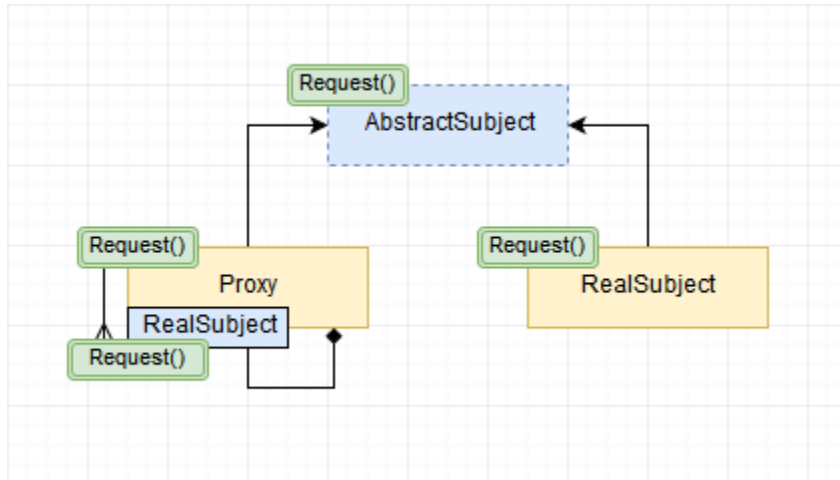
```
BaseAbstraction Flexible = new RefinedAbstraction();
Flexible.Implementation = new ImplementationA();
Flexible.Operation();
Flexible.Implementation = new ImplementationB();
Flexible.Operation();
```

PROXY (structural)

MEMORY HELPER: Gatekeeper wrapper

PATTERN OBJECTIVE: Provide a surrogate or placeholder for another object to control access to it.

EXAMPLE USAGE: Stand-in object for network resources that might be null at runtime. Access controlling a sensitive system object. Control/defer instantiation of an object.



BRIEF DESCRIPTION: The Proxy and RealSubject inherit from the same interface/abstract class to ensure uniformity. The client makes use of the RealSubject through the Proxy as follows:

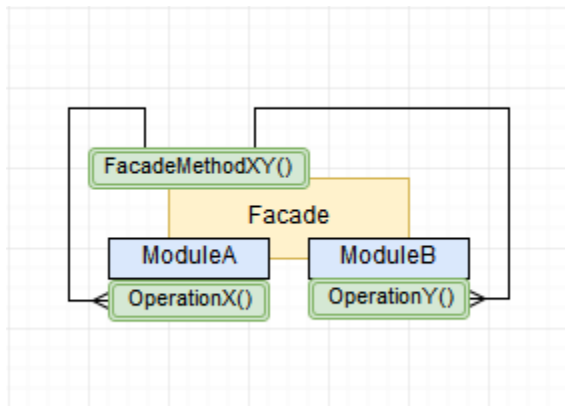
```
Subject proxy = new Proxy();
proxy.Request(); //calls RealSubject's Request
```

FACADE (structural)

MEMORY HELPER: Simplification via Composition

PATTERN OBJECTIVE: To provide a simplified class that reduces complexity of subsystem class modules that are brought together in Facade methods to complete more complex tasks. This helps decouple multiple dependencies by holding them in a single class while providing a simple way of “summarizing” a set of more complex tasks.

EXAMPLE USAGE: Creating a class that is composed of Inventory/Payment/Notification objects and calling their method in sequence in a single Facade Method:



BRIEF DESCRIPTION: The Facade class contains ModuleA and ModuleB subsystem instances and calls their operations through a single Facade Method:

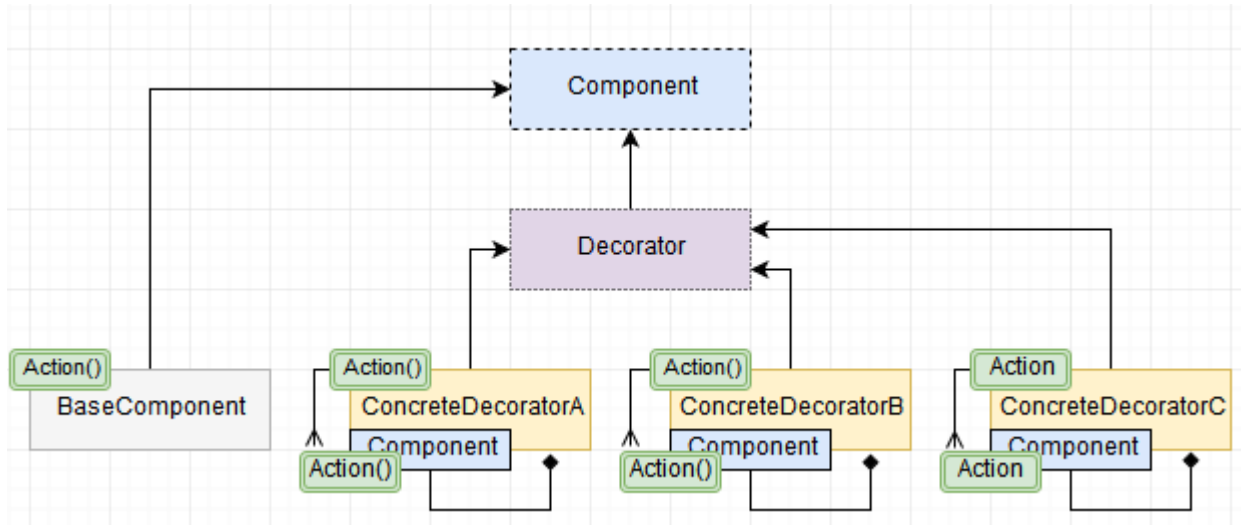
```
Facade facade = new Facade();
facade.FacadeMethodXY(); //Calls ModuleA.OperationX() and ModuleB.OperationY()
```

DECORATOR (structural)

MEMORY HELPER: Add object functionality dynamically

PATTERN OBJECTIVE: To add responsibilities to an object at runtime using composition rather than inheritance.

EXAMPLE USAGE: Creating a Window feature system (Borders, Shadows, Style, etc) as modular “decorators” that can be attached and removed easily.



BRIEF DESCRIPTION: Each ConcreteDecorator can contain a Component object (either a Base Component, which doesn’t attach any further Components, or another Decorator). When a Component’s Action() method is called, it cascades down the chain of attached Components and calls each Component’s Action() method, essentially acting a single unified lego-block object:

```
BaseComponent baseComponent = new BaseComponent();
```

```
Decorator firstDecoration = new ConcreteDecoratorA();  
firstDecoration.Component = baseComponent
```

```
Decorator secondDecoration = new ConcreteDecoratorB();  
secondDecoration.Component = firstDecoration;
```

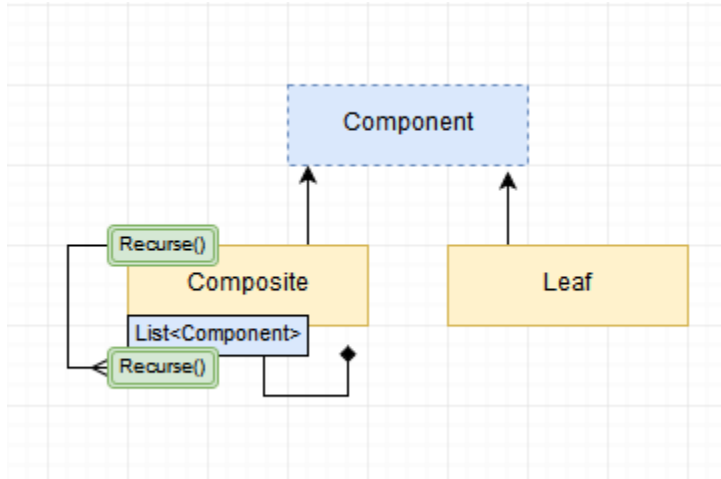
```
//Do secondDecoration, firstDecoration and baseComponent Action()  
secondDecoration.Action();
```

COMPOSITE (structural)

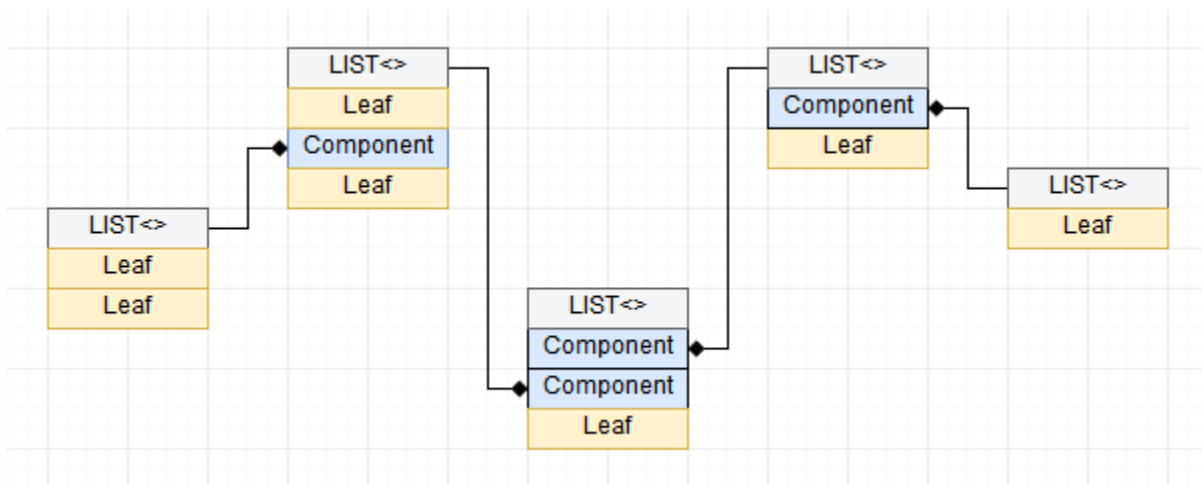
MEMORY HELPER: Object composition trees

PATTERN OBJECTIVE: To create a tree-like composite of objects that can be treated as individual parts or as grouped composites.

EXAMPLE USAGE: A Component-based Framework where you can add and remove components in a non-linear manner creating Composite objects (EG: GameObject/Components in Unity).



BRIEF DESCRIPTION: A Composite is very similar in principle to a Decorator except a Decorator's Component tree is linear - a Component can hold only another single Component - whereas a Composite's Component can hold any number of Component's within its **List<Component>**. A Leaf represents a "terminating" Component as it cannot have Components attached to it. You can recursively call each Component's methods in the **List<Component>** with a simple **foreach()** loop. The **List<Component>** tree can be visualized as follows:

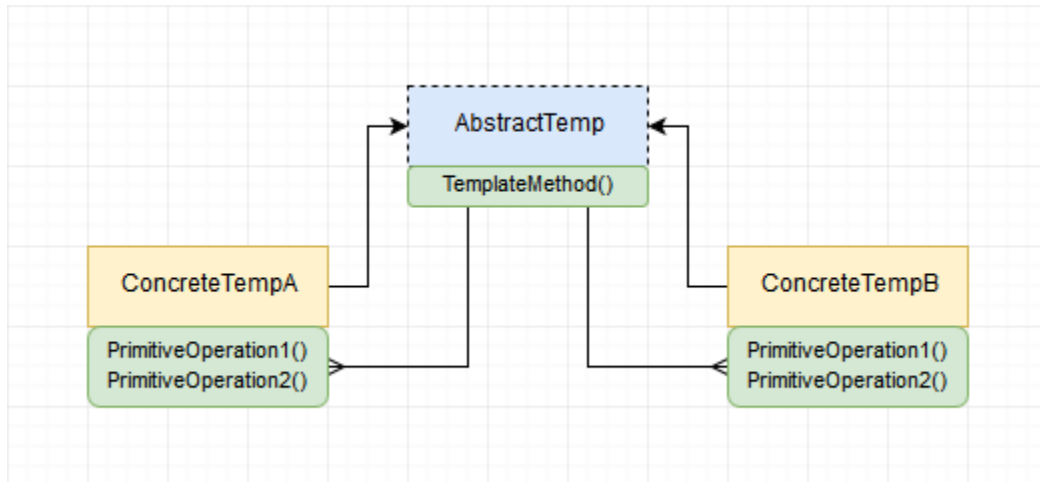


TEMPLATE METHOD (behavioral)

MEMORY HELPER: Default method call sequence

PATTERN OBJECTIVE: To specify a method that calls a common sequence of sub-methods, while allowing subclasses to redefine some parts of the sequence.

EXAMPLE USAGE: Encapsulate a sequence of common procedures in a single Template method, such as open/read/close (IE: a database query routine of opening a DB connection, selecting from the database, and closing the connection).



BRIEF DESCRIPTION: The ConcreteTempA and ConcreteTempB classes might hold a different way of committing a set of procedures, such as processing data. The TemplateMethod() inverts control since it comes from a parent class, calling the procedure set as a single sequence without knowing how the implementations differ:

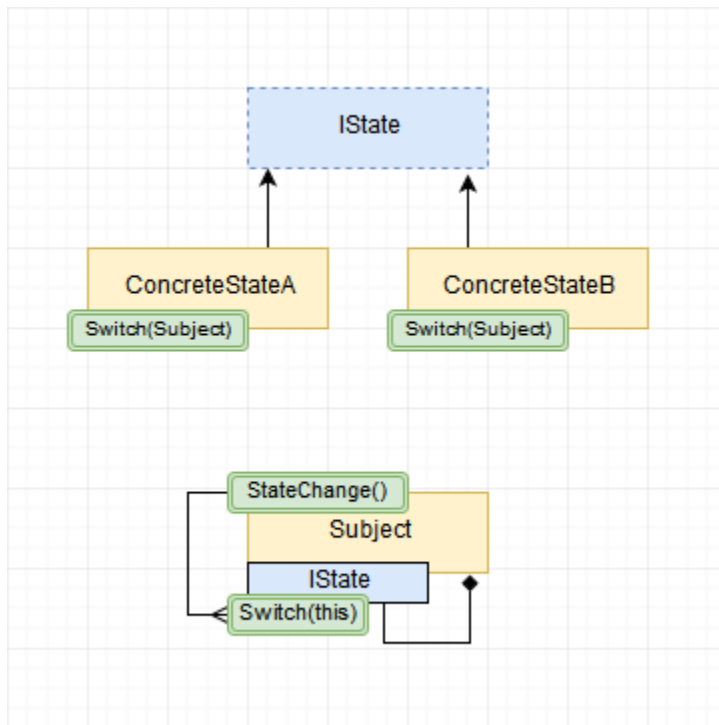
```
AbstractTemp a = new ConcreteTempA();
a.Sequence(); //parent calls ConcreteTempA's PrimitiveOperation's
AbstractTemp b = new ConcreteTempB();
b.Sequence(); //parent calls ConcreteTempB's PrimitiveOperation's
```


STATE (behavioral)

MEMORY HELPER: Change object behavior

PATTERN OBJECTIVE: Delegate an object's behavior to an internal composition object, allowing behaviors to be swapped out seamlessly. This can have the appearance of the object changing its type at runtime.

EXAMPLE USAGE: House different implementations for a "Subject" depending on its state within a State object held via Composition (EG: a video game character that controls differently when it transforms from one state to another).



BRIEF DESCRIPTION: The Subject has a concrete instance of IState, when something happens in the Subject that might trigger a state change, the Subject feeds itself into its IState Switch() method and let's the IState object's Switch() method determine if it should change to another instance of IState.

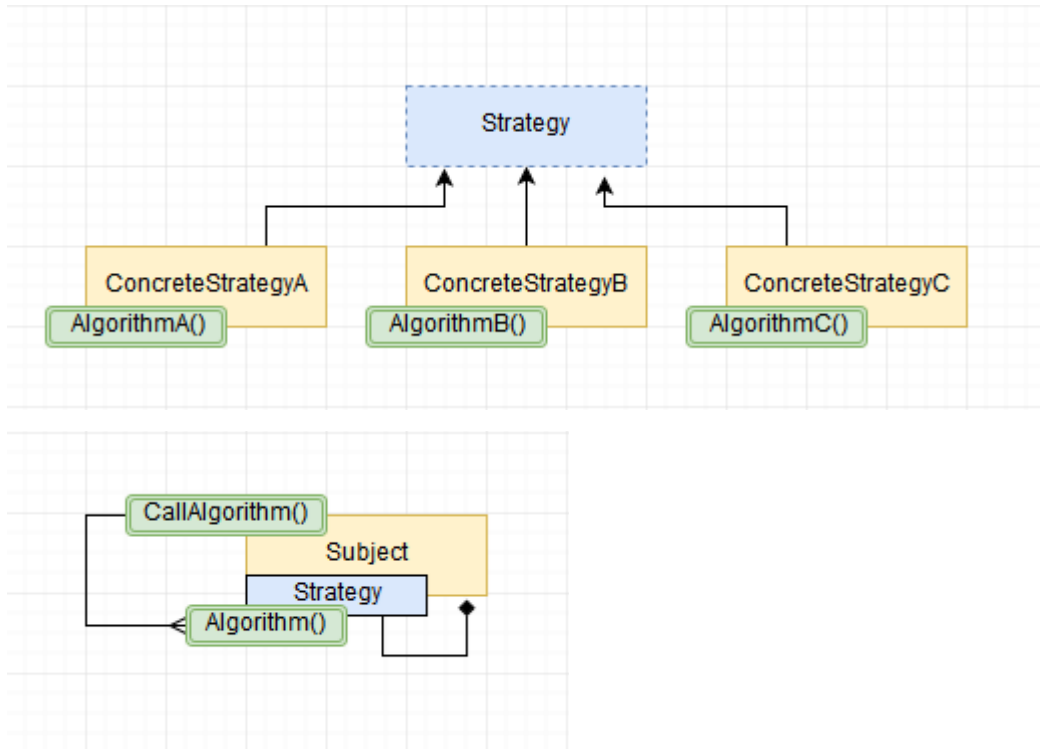
```
Subject subject = new Subject();
subject.StateChange(); //Subject.IState = ConcreteStateA
subject.StateChange(); //Subject.IState = ConcreteStateB
```

STRATEGY (behavioral)

MEMORY HELPER: Set class methods via composition object.

PATTERN OBJECTIVE: To delegate a similar group of method behaviors to outside objects.

EXAMPLE USAGE: House different implementations for a “Subject” within a Strategy object held via Composition. Similar to a State pattern, however lacking the ability to switch states, intending instead to change a “strategy” algorithm for completing similar tasks (EG: changing sorting strategies for a record-keeping application).



BRIEF DESCRIPTION: Unlike a State pattern, the Subject doesn’t feed itself into its Strategy object, and the Strategy is not responsible for changing the Subject’s strategy. The A/B/C algorithm’s for each concrete strategy represent different ways of approaching a problem (EG: SortByID, SortByName, SortBySalary):

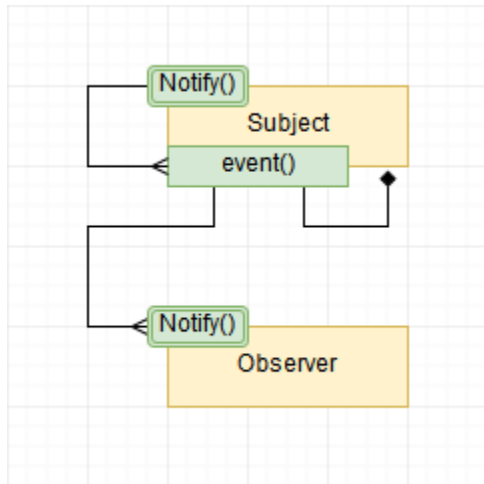
```
Subject subject = new Subject();
subject.strategy = new ConcreteStrategyB();
subject.CallAlgorithm(); //calls ConcreteStrategy.AlgorithmB()
```

OBSERVER (behavioral)

MEMORY HELPER: Subject notifies observers (events)

PATTERN OBJECTIVE: A system in which an object can alert its dependencies upon a condition being met. The dependencies can carry out some function in response to the notification.

EXAMPLE USAGE: This pattern is built into the C# language with its use of events. This is ideal for any point in application where a group of objects (Observers) need to be notified of something that's changed in the Subject and perhaps do something in response.



BRIEF DESCRIPTION: Any number of Observer's methods can subscribe to Subject's event object so long as they match the event's delegate method signature. When the event fires it calls those subscribed methods in all the observers.

```
Subject subject = new Subject();
Observer observer = new Observer();

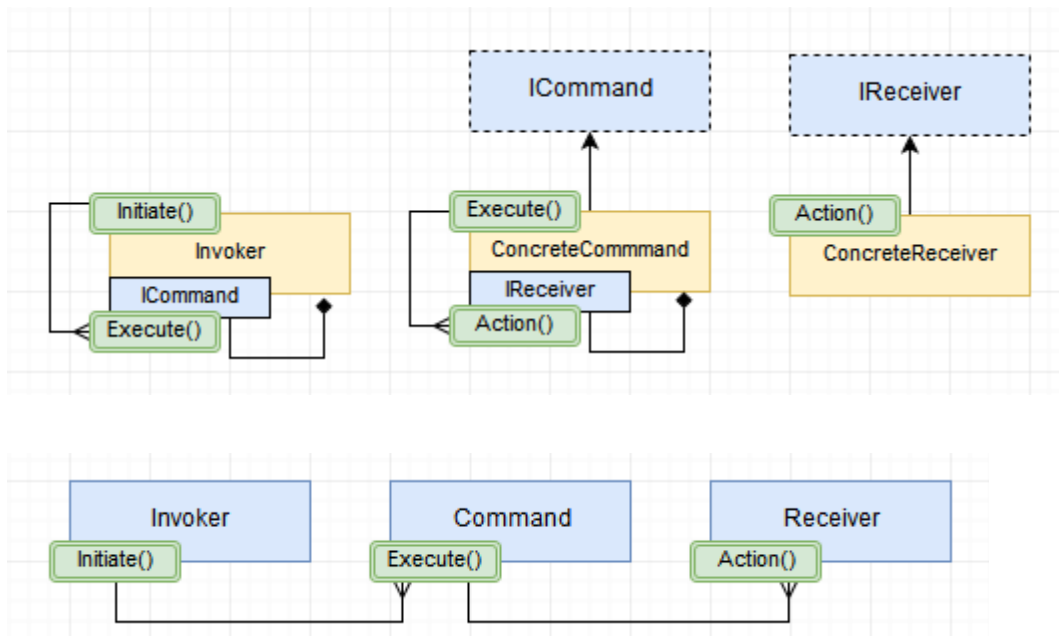
subject.EventHandler += observer.Notify;
subject.Notify(); //calls observer.Notify()
```

COMMAND (behavioral)

MEMORY HELPER: Encapsulate actions in an object

PATTERN OBJECTIVE: To create abstract representations of command-like behaviors (Command class) that can be separated from classes that act upon those commands (Receiver class). This can provide a layer of abstraction in a system that doesn't know precisely what kind of commands it will have to process. Also, by parceling commands into objects, they can be catalogued in interesting ways such as logging a history of actions or allowing undo/redo procedures.

EXAMPLE USAGE: Parceling command-like information that a receiving class can act upon. Holding the history of Command objects, such as `List<ICommand>`, in the Invoker allows you to undo the Command, log them and so on. A calculator app is a good example of this usage, where arithmetic operations can be contained in Command objects and later perform the opposite arithmetic procedure to undo the resulting calculation in the Receiver.



Invoker ---initiates--> **Command** ---sends-to--> **Receiver**

BRIEF DESCRIPTION: The Invoker class stores a Command - or `List<Command>`. The Invoker calls the Command's `Execute` method in its `Initiate()` method. Then the Command sends its information/itself to the Receiver by calling the Receiver's `Action()` method in its `Execute` method.

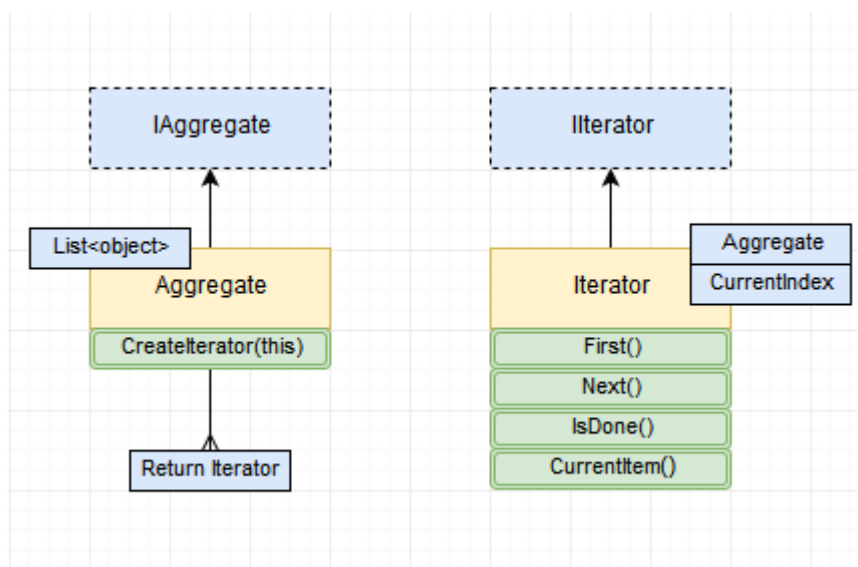
```
Invoker invoker = new Invoker();
invoker.SetCommand(new ConcreteCommand(new Receiver()));
invoker.InitiateCommand();
```

ITERATOR (behavioral)

MEMORY HELPER: Separate Iteration Procedures from a Collection

PATTERN OBJECTIVE: To allow highly customizable iteration behavior against aggregate collections (Arrays, ArrayLists, Lists, etc) by housing the aggregate data separate from its iteration procedures. The use of interfaces to represent the aggregate data as well as the iteration behaviors allows for flexible and modular approach to iterating over collections.

EXAMPLE USAGE: Any situation that involves iterating over collections of data. Applies well as a custom iteration scheme, handling large collections of data without losing the index left off on the previous iteration, or other scenarios where flexibility is key (EG: establish procedures for unknown future collection types).



BRIEF DESCRIPTION: Typical usage is to create an `Aggregate` instance in a client class and call the `CreateIterator()` method, passing in the `Aggregate` to the `Iterator` and returning the `Iterator`. You should rely on the `Iterator` to handle all iteration procedures rather than directly modifying the `Aggregate` (using `foreach()` loops or other procedures that aren't housed in the `Iterator`):

```
IAggregate aggregate = new Aggregate(  
    new object[] { "Item1", "Item2", "Item3", "Item4" }  
);  
  
IIterator iterator = aggregate.CreateIterator();  
  
Console.WriteLine(iterator.CurrentItem());
```