**TopCoder Thrive – Author: Yog-Sothoth – Track: Development – 5 min read**

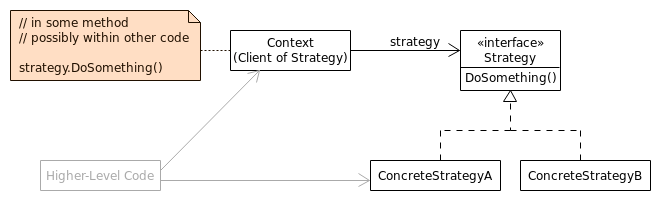
**Tags: Design Patterns, Strategy, C#, JavaScript**

**The Strategy Pattern in C# and JavaScript**

# INTRODUCTION

The Strategy pattern is one of the original patterns described in the famous [Design Patterns](https://www.amazon.com/Design-Patterns-Object-Oriented-Addison-Wesley-Professional-ebook/dp/B000SEIBB8) book. It is a so-called behavioral pattern, which means that it is a way to organize your code so that its behavior (what it actually does when executed) can be made flexible. This pattern in particular allows for details of a specific generalized behavior to be dynamically swappable, and decouples those details from the code that may use them so that the two can evolve independently. In other words, if there are different versions of the same algorithm or process (that may differ in implementation detail, but conceptually do the same thing), then the Strategy pattern may be a way to go.

# STRUCTURE



The core element of the Strategy pattern is the *Strategy abstraction*; in the image above, it is represented by an interface type, but it may also be an abstract class (what’s preferred may differ between languages). It defines a generalized interface (a set of public methods and properties) for different concrete strategies to implement. The code in the Context is written entirely in terms of this interface, so the interface must be made expressive enough to support anything that Context may want to do. (It may take several iterations to get this right.) In [duck-typed](https://en.wikipedia.org/wiki/Duck_typing) languages, the Strategy abstraction will often not be represented in code directly; instead it may be expressed as a requirement (e.g., in the documentation of the Context) that the object passed in to serve as a strategy must support a certain set of methods and/or properties (since those will be called/used in the Context). If the strategy boils down to a single method, it can be a reference to a function instead (a function pointer, a delegate, a lambda).

Some *other* code (the “Higher-Level Code” in the image) configures the Context with a ConcreteStrategy, typically via injection (the Context stores the strategy in a member variable that has the abstract Strategy type). Alternatively, if it’s a one-off thing, the strategy may be passed to a method as a parameter. This other code that composes the Context and a ConcreteStrategy is presumably written at a place in the codebase where, and at a time when, there is enough information for a specific ConcreteStrategy to be chosen. The Context and the Strategy can then work together to realize the desired behavior. Once a concrete strategy has been chosen, any other code that needs to make use of this behavior will only interact with the Context. When the Context receives a request (when a method is called on it), it will itself execute the general parts of the behavior, and forward to the ConcreteStrategy when it comes to details.

Encapsulating the details in this way will likely make the code of the Context simpler, as well as the code that calls it. Since clients don’t depend on those details, some of the concrete strategies can be added or accounted for later on, as the need arises.

# EXAMPLES

To make things more concrete, here are some examples of the Strategy pattern in C# and JavaScript.

## Let’s Start With C#

The .Net standard library itself makes use of the pattern in various ways. The behavior or the algorithm represented by the strategy doesn’t have to be a big part of the Context object; it may be some small aspect of it that depends on details or rules unknown to the Context. Take, for example, the often used List<T> class. It has a Sort method that performs an in-place sort. There is a default implementation, but there’s also an overload which takes as a parameter an *element comparison strategy* - an object that implements the IComparer<T> interface. This allows List<T> and its Sort method (the Context) to implement the sort algorithm without knowing how to actually compare the elements of a list; this part of the logic is forwarded to the comparer. For example, suppose the following list has to be sorted:

var shapes = new List<Shape> { rectangle, triangle, square, circle, pentagon };

The requirement is to sort these shapes by the number of sides they have (with circle is taken to have zero sides).

The IComparer<T> interface has a single method that takes two elements of type T and returns an integer value indicating their relative sort order. For two values x and y, a result that is < 0 means that x comes before y, 0 means they are of equal “rank”, while a value > 0 means that y comes before x. Note that this information is enough for the sort algorithm to do its job. A comparer that works with the NumSides property of the Shape class may look like this:

public class ShapeComparer : IComparer<Shape>   
{  
 public int Compare(Shape x, Shape y)  
 {  
 return x.NumSides - y.NumSides;  
 }  
}

An instance of the ShapeComparer can be passed to the sort method:

shapes.Sort(new ShapeComparer());   
// the list is now ordered like so:   
// circle, triangle, rectangle, square, pentagon

Since in many cases (including this one) a full blown class is not necessary, there’s another overload of Sort that takes a Comparison *delegate*, letting the callers pass in a simple lambda:

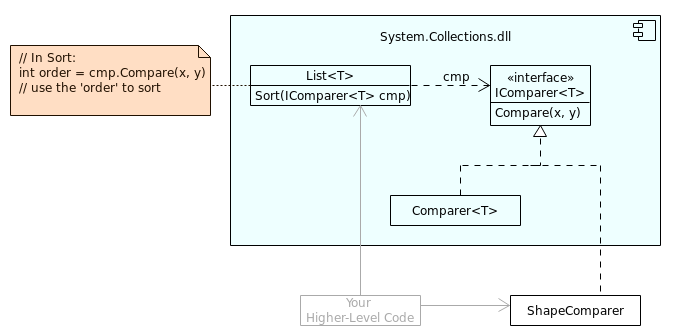
shapes.Sort((x, y) => x.NumSides - y.NumSides); // same as before  
shapes.Sort((x, y) => x.Name.Length - y.Name.Length); // sort by name length

This is a variation of the Strategy pattern, where the ConcreteStrategy is represented by a lambda expression, while the abstract Strategy “interface” is the delegate, which defines a certain signature. Another place where this same pattern is extensively used is LINQ. The IEnumerable<T> extension methods provide various ways to filter and transform collections, but many of them take one or more lambdas that let the underlying LINQ implementation be independent of the details of the actual elements, and any specialized logic that may be associated with them:

var q = shapes.Where(shape => shape.NumSides > 3).Select(shape => shape.Name);  
return string.Join(", ", q); // returns “rectangle, square, pentagon”

In the example above, the Where method takes in a *predicate*, which is a filtering strategy that tells it which elements to keep. The Select method takes a selector, that tells it how to transform (or “project”, or “map”) each element. Each of the context methods performs additional, more general logic that allows the result to be lazily evaluated and to behave like any other enumerable (e.g., it can work with foreach, and other LINQ methods, etc.).

Another thing to note here is that the separation of concerns achieved by the Strategy pattern allows for the Context and the Strategy to be placed in *different compilation units* (separate EXEs or DLLs).



While these examples have served to demonstrate how the .Net Framework itself makes use of the pattern, you can leverage it to organize code within your own projects. Consider a requirement to support serializing a business object into a couple of different formats. You can achieve this using the Strategy pattern. Furthermore, new formats can easily be supported in the future, and if you are using a plugin architecture, new serialization strategies can be deployed independently as plugins, eliminating the need to recompile the application.

// manual, hardcoded  
customerInfo.Serialize(new CustomerJsonSerializer());  
customerInfo.Serialize(new CustomerBinarySerializer());  
customerInfo.Serialize(new CustomerLogSerializer());  
  
// or via a factory based on user input  
customerInfo.Serialize(userSelectedSerializer);

## Examples in JavaScript

JavaScript is an interesting multiparadigm language. It is, among other things, object-oriented, and uses [prototype-based inheritance](https://developer.mozilla.org/en/docs/Web/JavaScript/Inheritance_and_the_prototype_chain), but it supports some functional features as well. It is also dynamic, and supports [duck-typing](https://en.wikipedia.org/wiki/Duck_typing). Similar to the .Net LINQ examples discussed above, JavaScript has Array processing methods, some of which make use of the strategy pattern. For example, there are [filter](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/filter), which takes a predicate, [reduce](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/Reduce), which combines all the elements into a single value using a caller supplied reduction strategy, and [map](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/map), which produces a new array by transforming each element using an external transformation function.

let sumReducer = (acc, current) => acc += current;  
let mulReducer = (acc, current) => acc \*= current;  
  
let mapping = ['a', 'b', 'c', 'd', 'e'];  
lookupReducer = (acc, current) => acc += mapping[current - 1];  
  
[1, 2, 3, 4, 5].reduce(sumReducer); // 15  
[1, 2, 3, 4, 5].reduce(mulReducer); // 120  
[1, 2, 3, 4, 5].reduce(lookupReducer, '') // "abcde"

The reduce method encodes the higher-level reduction algorithm, which involves iterating through the array, maintaining the accumulator variable, and some other rules (like the ones about the selection of the initial value, etc.). The step of actually combining the current item with the accumulator is forwarded to the reducer function. Note that for this to work the reducers all have to support a certain signature (a functional equivalent of the Strategy interface).

A Strategy can also be a composite thing, where several objects collaborate to achieve a result; in the slightly more complicated example below, the lamba works together with the PairCollection class to group the items into pairs:

class PairCollection {  
 constructor() {  
 this.pairs = [];  
 }  
  
 add(item) {  
 this.\_getLastGroup().push(item);  
 return this; // needed in order to be used as an accumulator  
 }  
   
 \_getLastGroup() {  
 if (this.pairs.length == 0 ||  
 this.pairs[this.pairs.length - 1].length == 2) {  
 this.pairs.push([]);  
 }  
 return this.pairs[this.pairs.length - 1];  
 }  
}  
[1, 2, 3, 4, 5].reduce((acc, current) => acc.add(current), new PairCollection())  
// produces: [ [ 1, 2 ], [ 3, 4 ], [ 5 ] ]

Here’s another example. Suppose there is a slideshow or a carousell on a page, and that it should be able to auto-advance slides in different ways (sequentially, in random order, etc.). If the slideshow is managed by a SlideshowController that can be configured by a slide selection strategy, then it is easy to dynamically change the way auto-advance behaves:

class SlideshowController {  
 constructor(slidesArray, selectionStrategy) {  
 this.\_slidesArray = slidesArray;  
 this.\_currentSlideIndex = 0;  
 this.selectionStrategy = selectionStrategy;  
 }  
  
 getNextSlide() {  
 this.\_currentSlideIndex = this.selectionStrategy(  
 this.\_currentSlideIndex, this.\_slidesArray.length);  
 return this.\_currentSlideIndex;  
 }  
  
 // other methods omitted…  
}

The code that controls the containing page can define a couple of slide selection strategies, and inject them into the SlideshowController as a response to an event resulting from a user action:

this.\_defaultSelectionStrategy = (current, count) => (current + 1) % count;

this.\_randomSelectionStrategy = (current, count) =>   
 Math.floor(Math.random() \* count);

this.\_slideshowController = new SlideshowController(  
 slides, defaultSelectionStrategy);

$('.btn-sequential').click(() => this.\_slideshowController.selectionStrategy  
 = this.\_defaultSelectionStrategy);

$('.btn-random').click(() => this.\_slideshowController.selectionStrategy  
 = this.\_randomSelectionStrategy);

Another common use for the pattern is for testing; strategies can serve as mocks. If the code under test relies on a service of some kind, but it is inconvenient to use the actual service during everyday development (e.g., it makes a remote call), the service can be substituted for a mock object that supports the same interface, but returns mock data, that can be made specific to a particular test if necessary. This works well with [dependency injection](https://en.wikipedia.org/wiki/Dependency_injection).

## Summary

The Strategy pattern achieves decoupling by encapsulating variable parts of a behavior behind an abstract interface. This abstraction can make working with a family of related algorithms easier, and it lets you postpone introduction of new ConcreteStrategies. The structural aspects of Strategy allow for ConcreteStrategies to be dynamically swappable, and even to be packaged and deployed separately. As exemplified by the use of the pattern in duck-typed languages, the abstract Strategy interface can be entirely conceptual (i.e., not explicitly defined in code), and still result in decoupling, because programming to an interface rather than to an implementation is first and foremost a design decision. Of course, this requires some programmer discipline, as well as good documentation, and a very careful consideration of what that conceptual interface should look like. In statically typed languages, on the other hand, the compiler can help the programmer by doing type checking; it is still necessary, however, to understand the semantics of the abstract Strategy interface in order to implement ConcreteStrategies properly.