
Behavioral Patterns: Strategy, State, and Template Method

Behavioral patterns are concerned with algorithms and communication between them. The operations that make up a single algorithm might be split up between different classes, making a complex arrangement that is difficult to manage and maintain. The behavioral patterns capture ways of expressing the division of operations between classes and optimize how the communication should be handled. In this first chapter on behavioral patterns, we'll look at three simple but very useful patterns: the Strategy, State, and Template Methods.

Strategy Pattern

Role

The Strategy pattern involves removing an algorithm from its host class and putting it in a separate class. There may be different algorithms (strategies) that are applicable for a given problem. If the algorithms are all kept in the host, messy code with lots of conditional statements will result. The Strategy pattern enables a client to choose which algorithm to use from a family of algorithms and gives it a simple way to access it. The algorithms can also be expressed independently of the data they are using.

Illustration

A classic illustration of the Strategy pattern is found in the choice of algorithms for sorting. There are many sorting algorithms, and although some, such as Quicksort, are generally very fast, there are situations when this would be a poor choice and another algorithm, such as Mergesort, would perform better. Even the linear sorts, such as Shellsort, can perform very well under certain conditions. When studying sorting, one learns about the different conditions to consider and how to optimize the choice of algorithms. This is a strategy that can be captured in the Strategy pattern.

Sorting lends itself to animation, as shown in Figure 7-1. The graph plots the index against the value at successive stages in the sort. Thus, initially the box shows a scatter of points. As the algorithm progresses, the points start to converge on the diagonal, indicating that the values are in the correct positions in the list. The lefthand window shows Mergesort in action, and the righthand window shows Quicksort.

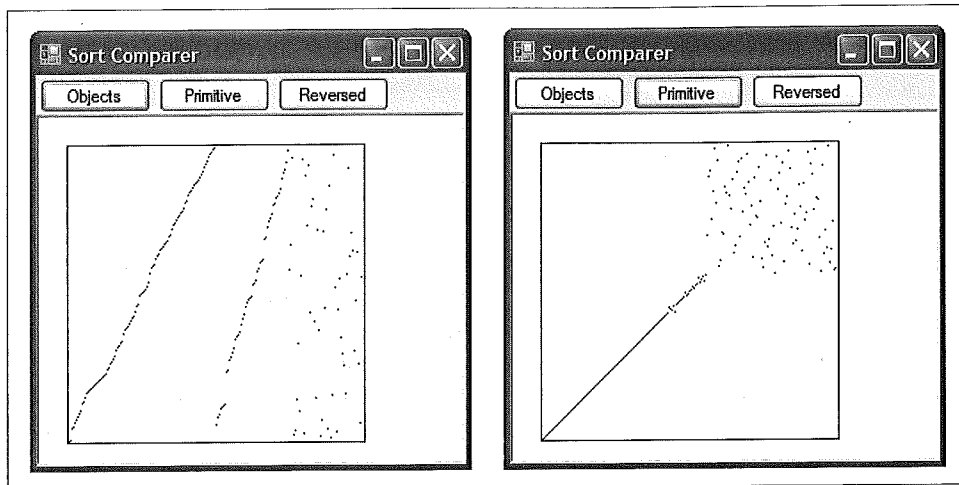


Figure 7-1. Strategy pattern illustration—sorting objects with Mergesort and sorting primitive types with Quicksort

In terms of strategy, the animator is set up so that the user indicates the type of items being sorted (at a very rough level in this example). The underlying Strategy pattern then selects an appropriate sort method. In this case, larger values (objects) that have costly compare operations are sent to Mergesort, which uses the lowest number of comparisons of all popular sorts. On the other hand, Quicksort can go very fast with primitive items (where the comparisons are cheap), so it is preferred for those types of values. The third button is there because of a limitation of Quicksort: without careful programming, it is very slow with reversed data. This button can be used to activate Mergesort. The Strategy pattern makes it easy to add other criteria and sorts to the animator as needed.

Design

The design of the Strategy pattern is encapsulated in the UML diagram in Figure 7-2. Within a given Context, an appropriate strategy is chosen from an available family of strategies. The algorithm in the strategy is then followed through to a conclusion.

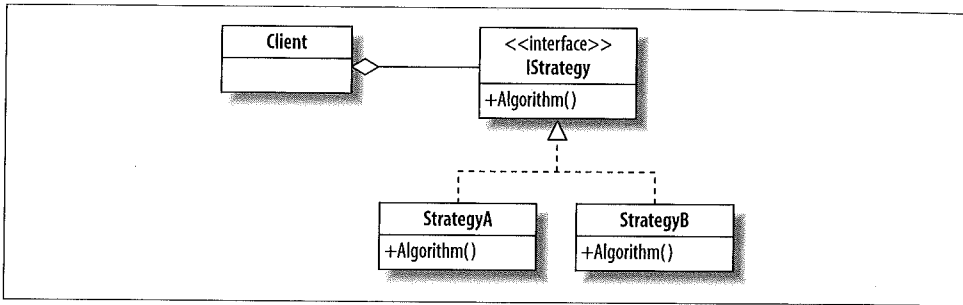


Figure 7-2. Strategy pattern UML diagram

The roles for the players in this pattern are as follows:

Context

A class that maintains contextual information for an IStrategy object's algorithm to work on

IStrategy

Defines an interface common to all the strategies

StrategyA, StrategyB

Classes that include algorithms that implement the IStrategy interface

QUIZ

Match the Strategy Pattern Players with the Sorting Animator Illustration

To test whether you understand the Strategy pattern, cover the lefthand column of the table below and see if you can identify its players among the items from the illustrative example (Figure 7-2), as shown in the righthand column. Then check your answers against the lefthand column.

| | |
|-----------|---|
| Context | The GUI, the list generator, and the selection of the Strategy |
| Strategy | Class containing the methods used for a particular sort |
| IStrategy | Sorting interface (e.g., specifying the list and its item type) |
| Algorithm | Mergesort or Quicksort |

As defined, the Context aggregates an object of the chosen Strategy type. Generally, it will only have access to the main method that the algorithm requires. If it needs more information from the Strategy, this should be included in the IStrategy interface. On the other hand, the Strategy will need to work within the Context and access its state.

Implementation

The theory code for the Strategy pattern (Example 7-1) does not need any new C# 3.0 features. It relies on aggregation of the IStrategy interface in the Context (line 13). The client calls the Algorithm method on the Context (line 59), and it is routed through to the method of the strategy applicable at the time. In this example, the strategies have Move methods that count up and down. A random number generated in the client determines when to switch from counting up to counting down. The resulting output is shown in line 65.

Example 7-1. Strategy pattern theory code

```
1   using System;
2
3   // Strategy Pattern           Judith Bishop Oct 2007
4   // Shows two strategies and a random switch between them
5
6   // The Context
7   class Context {
8       // Context state
9       public const int start = 5;
10      public int Counter = 5;
11
12      // Strategy aggregation
13      IStrategy strategy = new Strategy1();
14
15      // Algorithm invokes a strategy method
16      public int Algorithm() {
17          return strategy.Move(this);
18      }
19
20      // Changing strategies
21      public void SwitchStrategy() {
22          if (strategy is Strategy1)
23              strategy = new Strategy2();
24          else
25              strategy = new Strategy1();
26      }
27  }
28
29  // Strategy interface
30  interface IStrategy {
31      int Move (Context c);
32  }
33
```

Example 7-1. Strategy pattern theory code (continued)

```
34 // Strategy 1
35 class Strategy1 : IStrategy {
36     public int Move (Context c) {
37         return ++c.Counter;
38     }
39 }
40
41 // Strategy 2
42 class Strategy2 : IStrategy {
43     public int Move (Context c) {
44         return --c.Counter ;
45     }
46 }
47
48 // Client
49 static class Program {
50     static void Main () {
51         Context context = new Context();
52         context.SwitchStrategy();
53         Random r = new Random(37);
54         for (int i=Context.start; i<=Context.start+15; i++) {
55             if (r.Next(3) == 2) {
56                 Console.Write("|| ");
57                 context.SwitchStrategy();
58             }
59             Console.Write(context.Algorithm() + " ");
60         }
61         Console.WriteLine();
62     }
63 }
64 /* Output
65 4 || 5 6 7 || 6 || 7 8 9 10 || 9 8 7 6 || 7 || 6 5
66 */
```

Some key points about the implementation of the Strategy pattern are:

- The Context will often contain a switch statement or a cascading if statement, where information is processed to reach a decision on which Strategy to adopt.
- If the strategies are simple methods, they can be implemented without enclosing classes, and the delegate mechanism can be used to hook the chosen Strategy into the Context at runtime.
- Extension methods can be used to define new strategies independently of the original classes that they support.

Example: Sorting Animator

The program that produces the animations in Figure 7-1 is shown in Example 7-2. Consider first the Context (lines 61–86). `ButtonClick` is activated from the GUI and, based on the button clicked, will decide on a strategy to follow. In other words, it will select one of the available strategy classes and instantiate it. After generating data from the class at line 15 (not shown in full here), it activates the GUI window and starts the sort. We don't show the full sorting algorithms here, only the interaction with the Context (see lines 96–112 for Mergesort). Input is the list that is being sorted, and the algorithms contain strategic calls to update the user interface through the `UpdateUI` event.

Example 7-2. Strategy pattern example code—Sorting Animator

```
1   using System;
2   using System.Collections.Generic;
3   using System.Linq;
4   using System.Windows.Forms;
5   using System.Drawing;
6   using System.Threading;
7
8   namespace Strategy {
9
10      // Strategy Pattern      Judith Bishop and D-J Miller  Sept 2007
11      // Gives a choice of sort routines to display
12      // Algorithms and GUI adapted from a Java system at
13      // http://www.geocities.com/SiliconValley/Network/1854/Sort1.html
14
15      static class StartSetGenerator {
16          private static List<int> myList;
17
18          public static IEnumerable<int> GetStartSet() {
19              // omitted
20          }
21
22      class StrategyView<T> : Form
23          where T : IComparable<T> {
24          PictureBox pb;
25          Func<IEnumerable<T>> Generator;
26
27          // Constructor to set up the GUI
28          public StrategyView(Func<IEnumerable<T>> generator) {
29              // omitted
30
31          public void DrawGraph(IEnumerable<T> list) {
32              if (pb.Image == null)
33                  pb.Image = new Bitmap(pb.Width, pb.Height);
34              Graphics g = Graphics.FromImage(pb.Image);
35              g.Clear(Color.White);
36              g.DrawRectangle(Pens.Blue, 19, 19, 202, 202);
37              g.Dispose();
38              Bitmap b = pb.Image as Bitmap;
39
```

Example 7-2. Strategy pattern example code—Sorting Animator (continued)

```
40     // Plots the index x against the value val of all elements in the list
41     // IEnumerable<T>.Count is an extension
42     int listSize = list.Count();
43     int x = 0;
44     foreach (T item in list) {
45         // val must be a nullable integer. The as operator will return null
46         // if it cannot convert the item to int
47         int? val = item as int?;
48         if (!val.HasValue)
49             val = 0;
50         // Drawing methods do not handle nullable types
51         b.SetPixel(x + 20, 20 + 200 - ((int)val), Color.Black);
52         x++;
53     }
54
55     this.Refresh();
56     Thread.Sleep(100);
57     Application.DoEvents();
58 }
59
60 // The Context
61 void ButtonClick(object sender, EventArgs e) {
62     Button control = sender as Button;
63     SortStrategy<T> strategy = null;
64
65     switch (control.Name) {
66         case "LargeItems":
67             strategy = new MergeSorter<T>();
68             break;
69         case "SmallItems":
70             strategy = new QuickSorter<T>();
71             break;
72         case "ReversedList":
73             strategy = new MergeSorter<T>();
74             break;
75     }
76
77     IEnumerable<T> newList = Generator();
78     DrawGraph(newList);
79     if (strategy == null)
80         return;
81
82     // DrawGraph will be invoked during sorting when
83     // the UpdateUI event is triggered
84     strategy.UpdateUI += new Action<IEnumerable<T>>(DrawGraph);
85     strategy.Sort(newList);
86 }
87 }
88
89 // Strategy interface
90 interface SortStrategy<T> where T : IComparable<T> {
91     event Action<IEnumerable<T>> UpdateUI;
```

Example 7-2. Strategy pattern example code—Sorting Animator (continued)

```
92     void Sort(IEnumerable<T> input);
93     }
94
95     // Strategy 1
96     class MergeSorter<T> : SortStrategy<T>
97         where T : IComparable<T> {
98
99         public event Action<IEnumerable<T>> UpdateUI;
100
101         List<T> aux;
102         int opCount = 0;
103         public void Sort(IEnumerable<T> input) {
104             UpdateUI(input);
105             opCount++;
106             List<T> sorteditems = new List<T>(input);
107             aux = new List<T>(sorteditems.Count);
108             for (int i = 0; i < sorteditems.Count; i++)
109                 aux.Add(default(T));
110             MergeSort(ref sorteditems, 0, sorteditems.Count - 1);
111             UpdateUI(sorteditems);
112         }
113
114         private void Merge(ref List<T> a, int l, int m, int r) {
115             // omitted
116
117             private void MergeSort(ref List<T> a, int l, int r) {
118                 // omitted
119             }
120
121             // Strategy 2
122             class QuickSorter<T> : SortStrategy<T>
123                 where T : IComparable<T> {
124
125                 public event Action<IEnumerable<T>> UpdateUI;
126
127                 int opCount = 0;
128                 public void Sort(IEnumerable<T> input) {
129                     UpdateUI(input);
130                     opCount++;
131                     List<T> sorteditems = new List<T>(input);
132
133                     QuickSort(ref sorteditems, 0, sorteditems.Count - 1);
134                     UpdateUI(sorteditems);
135                 }
136
137                 private int Partition(ref List<T> a, int l, int r) {
138                     // omitted
139
140                 private void QuickSort(ref List<T> a, int l, int r) {
141                     // omitted
142                 }
143
144                 static class Program {
```

Example 7-2. Strategy pattern example code—Sorting Animator (continued)

```
145     static void Main() {
146         Application.EnableVisualStyles();
147         Application.SetCompatibleTextRenderingDefault(false);
148         Application.Run(new StrategyView<int>(StartSetGenerator.GetStartSet));
149     }
150 }
151 }
```

UpdateUI is an event that is set in line 84 (in the Context) to refer to DrawGraph. This is a more transparent way of the Strategy getting back to the animator than having it call DrawGraph directly. Notice that DrawGraph (lines 31–58) has an interesting loop that uses a new feature in C# 2.0: *nullable types*.

C# Feature—Nullable Types

Nullable types add to primitive types and structs the ability to include an “undefined value.” In keeping with objects, the value is called null, and it can be assigned and checked for.

In programs, all primitive types are assigned default values on instantiation. Therefore, the ability to assign null to numeric and Boolean types is particularly useful when dealing with databases and other data types containing elements that may not be assigned a value. For example, a Boolean field in a database can store the values true or false, or it may be undefined.

In C#, a nullable type is declared with the addition of a question mark (e.g., int? x). An extra property—HasValue—can then be used to check whether a value is non-null. To convert back to a non-nullable type, use the as operator. as will return null if it cannot convert the value. For example, int? val = DateTime.Now as int?;

cf. C# Language Specification Version 3.0, September 2007, Section 4.1.10

Consider the loop in question. The items in the list are of type T, and it is possible that they might not have values. The cast to int on line 47 will therefore be trapped unless null is included as a possibility for val. In the next line, we can convert any possible nulls to zeros. Then, on line 51, we see that we have to convert the int? type to int so that it can be used in arithmetic:

```
44     foreach (T item in list) {
45         // val must be an integer. The as conversion needs it
46         // also to be a non-nullable, which is checked by the ?
47         int? val = item as int?;
48         if (!val.HasValue)
49             val = 0;
50         // Drawing methods do not handle nullable types
51         b.SetPixel(x + 20, 20 + 200 - ((int)val), Color.Black);
52         x++;
53     }
```

Use

There are many examples where a different algorithm can be passed to an active method or class. Dialog boxes, for example, can have different validating strategies depending on the kind of data required, whereas graph-drawing programs can accept algorithms for drawing pie charts, histograms, or bar charts. The C# 3.0 LINQ support libraries (for database queries) use this kind of separation between data and algorithm extensively.

Use the Strategy pattern when...

- Many related classes differ only in their behavior.
 - There are different algorithms for a given purpose, and the selection criteria can be codified.
 - The algorithm uses data to which the client should not have access.
-

Exercises

1. One of the criteria for choosing a sorting algorithm is whether the data is initially presorted, reverse-sorted, or in random order. Add to the Generator class in the Sorting Animator so that different arrangements of data in the list can be created. Then, add some more buttons to make the user's description of the problem at least two-dimensional (i.e., size of elements and arrangement of data). Change the strategy selection accordingly.
2. Add to the animator a linear sort like Shellsort, and define its criteria so that it can also be selected when appropriate. (Hint: Shellsort is good enough for short lists.)
3. In the proxy-protected version of MySpaceBook (Chapter 2), the registration process does not check whether the input is valid. Decide on more rich input (such as email addresses and requirements for passwords), and implement the input of each field using the Strategy pattern.

State Pattern

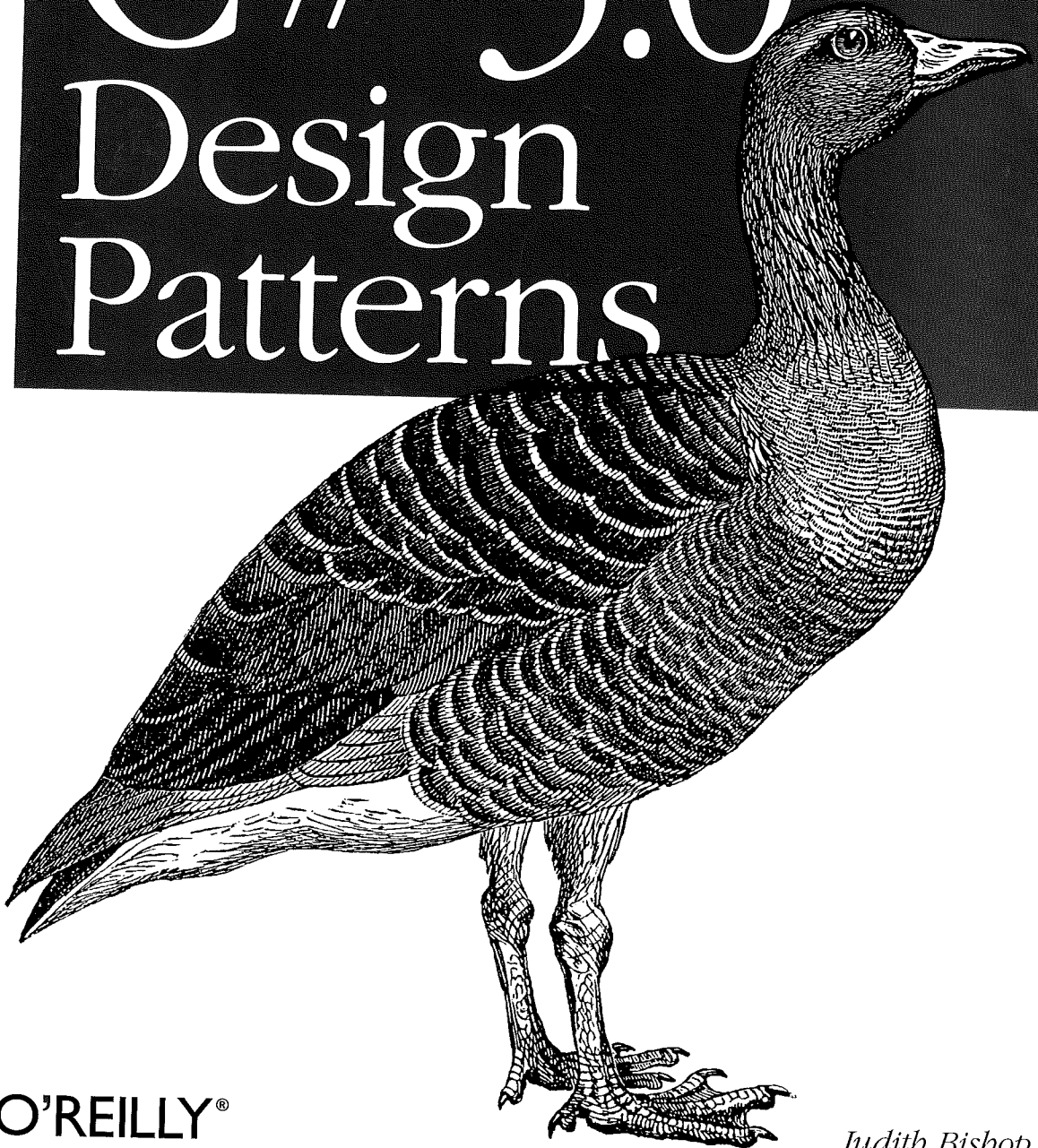
Role

The next pattern in this group, the State pattern, can be seen as a dynamic version of the Strategy pattern. When the state inside an object changes, it can change its behavior by switching to a set of different operations. This is achieved by an object variable changing its subclass, within a hierarchy.

*Use the Power of C# 3.0
to Solve Real-World Problems*

Up-to-date for C# 3.0

C# 3.0 Design Patterns



O'REILLY®

Judith Bishop