# Tutorial: Use pattern matching to build type-driven and data-driven algorithms

Article • 02/15/2023 • 15 minutes to read

You can write functionality that behaves as though you extended types that may be in other libraries. Another use for patterns is to create functionality your application requires that isn't a fundamental feature of the type being extended.

In this tutorial, you'll learn how to:

- ✓ Recognize situations where pattern matching should be used.
- ✓ Use pattern matching expressions to implement behavior based on types and property values.
- ✓ Combine pattern matching with other techniques to create complete algorithms.

## Prerequisites

- We recommend Visual Studio for Windows or Mac. You can download a free version from the Visual Studio downloads page. Visual Studio includes the .NET SDK.
- You can also use the Visual Studio Code editor. You'll need to install the latest .NET SDK separately.
- If you prefer a different editor, you need to install the latest .NET SDK.

This tutorial assumes you're familiar with C# and .NET, including either Visual Studio or the .NET CLI.

## Scenarios for pattern matching

Modern development often includes integrating data from multiple sources and presenting information and insights from that data in a single cohesive application. You and your team won't have control or access for all the types that represent the incoming data.

The classic object-oriented design would call for creating types in your application that represent each data type from those multiple data sources. Then, your application

would work with those new types, build inheritance hierarchies, create virtual methods, and implement abstractions. Those techniques work, and sometimes they're the best tools. Other times you can write less code. You can write more clear code using techniques that separate the data from the operations that manipulate that data.

In this tutorial, you'll create and explore an application that takes incoming data from several external sources for a single scenario. You'll see how **pattern matching** provides an efficient way to consume and process that data in ways that weren't part of the original system.

Consider a major metropolitan area that is using tolls and peak time pricing to manage traffic. You write an application that calculates tolls for a vehicle based on its type. Later enhancements incorporate pricing based on the number of occupants in the vehicle. Further enhancements add pricing based on the time and the day of the week.

From that brief description, you may have quickly sketched out an object hierarchy to model this system. However, your data is coming from multiple sources like other vehicle registration management systems. These systems provide different classes to model that data and you don't have a single object model you can use. In this tutorial, you'll use these simplified classes to model for the vehicle data from these external systems, as shown in the following code:

C#

```
namespace ConsumerVehicleRegistration
{
    public class Car
    {
        public int Passengers { get; set; }
    }
}

namespace CommercialRegistration
{
    public class DeliveryTruck
    {
        public int GrossWeightClass { get; set; }
    }
}

namespace LiveryRegistration
{
    public class Taxi
    {
```

```
        public int Fares { get; set; }
    }

    public class Bus
    {
        public int Capacity { get; set; }
        public int Riders { get; set; }
    }
}
```

You can download the starter code from the dotnet/samples GitHub repository. You can see that the vehicle classes are from different systems, and are in different namespaces. No common base class, other than `System.Object` can be used.

# Pattern matching designs

The scenario used in this tutorial highlights the kinds of problems that pattern matching is well suited to solve:

- The objects you need to work with aren't in an object hierarchy that matches your goals. You may be working with classes that are part of unrelated systems.
- The functionality you're adding isn't part of the core abstraction for these classes. The toll paid by a vehicle *changes* for different types of vehicles, but the toll isn't a core function of the vehicle.

When the *shape* of the data and the *operations* on that data aren't described together, the pattern matching features in C# make it easier to work with.

# Implement the basic toll calculations

The most basic toll calculation relies only on the vehicle type:

- A `Car` is $2.00.
- A `Taxi` is $3.50.
- A `Bus` is $5.00.
- A `DeliveryTruck` is $10.00

Create a new `TollCalculator` class, and implement pattern matching on the vehicle type to get the toll amount. The following code shows the initial implementation of the

`TollCalculator`.

C#

```
using System;
using CommercialRegistration;
using ConsumerVehicleRegistration;
using LiveryRegistration;

namespace Calculators;

public class TollCalculator
{
    public decimal CalculateToll(object vehicle) =>
        vehicle switch
    {
        Car c           => 2.00m,
        Taxi t          => 3.50m,
        Bus b           => 5.00m,
        DeliveryTruck t => 10.00m,
        { }             => throw new ArgumentException(message: "Not a known
vehicle type", paramName: nameof(vehicle)),
        null            => throw new ArgumentNullException(nameof(vehicle))
    };
}
```

The preceding code uses a switch expression (not the same as a switch statement) that tests the declaration pattern. A **switch expression** begins with the variable, `vehicle` in the preceding code, followed by the `switch` keyword. Next comes all the **switch arms** inside curly braces. The `switch` expression makes other refinements to the syntax that surrounds the `switch` statement. The `case` keyword is omitted, and the result of each arm is an expression. The last two arms show a new language feature. The `{ }` case matches any non-null object that didn't match an earlier arm. This arm catches any incorrect types passed to this method. The `{ }` case must follow the cases for each vehicle type. If the order were reversed, the `{ }` case would take precedence. Finally, the `null` constant pattern detects when `null` is passed to this method. The `null` pattern can be last because the other patterns match only a non-null object of the correct type.

You can test this code using the following code in `Program.cs`:

C#

```
using System;
```

```
using CommercialRegistration;
using ConsumerVehicleRegistration;
using LiveryRegistration;

using toll_calculator;

var tollCalc = new TollCalculator();

var car = new Car();
var taxi = new Taxi();
var bus = new Bus();
var truck = new DeliveryTruck();

Console.WriteLine($"The toll for a car is {tollCalc.CalculateToll(car)}");
Console.WriteLine($"The toll for a taxi is {tollCalc.CalculateToll(taxi)}");
Console.WriteLine($"The toll for a bus is {tollCalc.CalculateToll(bus)}");
Console.WriteLine($"The toll for a truck is
{tollCalc.CalculateToll(truck)}");

try
{
    tollCalc.CalculateToll("this will fail");
}
catch (ArgumentException e)
{
    Console.WriteLine("Caught an argument exception when using the wrong
type");
}
try
{
    tollCalc.CalculateToll(null!);
}
catch (ArgumentNullException e)
{
    Console.WriteLine("Caught an argument exception when using null");
}
```

That code is included in the starter project, but is commented out. Remove the comments, and you can test what you've written.

You're starting to see how patterns can help you create algorithms where the code and the data are separate. The `switch` expression tests the type and produces different values based on the results. That's only the beginning.

# Add occupancy pricing

The toll authority wants to encourage vehicles to travel at maximum capacity. They've decided to charge more when vehicles have fewer passengers, and encourage full vehicles by offering lower pricing:

- Cars and taxis with no passengers pay an extra $0.50.
- Cars and taxis with two passengers get a $0.50 discount.
- Cars and taxis with three or more passengers get a $1.00 discount.
- Buses that are less than 50% full pay an extra $2.00.
- Buses that are more than 90% full get a $1.00 discount.

These rules can be implemented using a property pattern in the same switch expression. A property pattern compares a property value to a constant value. The property pattern examines properties of the object once the type has been determined. The single case for a `Car` expands to four different cases:

C#

```
vehicle switch
{
    Car {Passengers: 0} => 2.00m + 0.50m,
    Car {Passengers: 1} => 2.0m,
    Car {Passengers: 2} => 2.0m - 0.50m,
    Car                 => 2.00m - 1.0m,

    // ...
};
```

The first three cases test the type as a `Car`, then check the value of the `Passengers` property. If both match, that expression is evaluated and returned.

You would also expand the cases for taxis in a similar manner:

C#

```
vehicle switch
{
    // ...

    Taxi {Fares: 0}  => 3.50m + 1.00m,
    Taxi {Fares: 1}  => 3.50m,
    Taxi {Fares: 2}  => 3.50m - 0.50m,
    Taxi             => 3.50m - 1.00m,
```

```
    // ...
};
```

Next, implement the occupancy rules by expanding the cases for buses, as shown in the following example:

C#

```
vehicle switch
{
    // ...

    Bus b when ((double)b.Riders / (double)b.Capacity) < 0.50 => 5.00m +
2.00m,
    Bus b when ((double)b.Riders / (double)b.Capacity) > 0.90 => 5.00m -
1.00m,
    Bus => 5.00m,

    // ...
};
```

The toll authority isn't concerned with the number of passengers in the delivery trucks. Instead, they adjust the toll amount based on the weight class of the trucks as follows:

- Trucks over 5000 lbs are charged an extra $5.00.
- Light trucks under 3000 lbs are given a $2.00 discount.

That rule is implemented with the following code:

C#

```
vehicle switch
{
    // ...

    DeliveryTruck t when (t.GrossWeightClass > 5000) => 10.00m + 5.00m,
    DeliveryTruck t when (t.GrossWeightClass < 3000) => 10.00m - 2.00m,
    DeliveryTruck => 10.00m,
};
```

The preceding code shows the `when` clause of a switch arm. You use the `when` clause to test conditions other than equality on a property. When you've finished, you'll have a method that looks much like the following code:

C#

```
vehicle switch
{
    Car {Passengers: 0}        => 2.00m + 0.50m,
    Car {Passengers: 1}        => 2.0m,
    Car {Passengers: 2}        => 2.0m - 0.50m,
    Car                        => 2.00m - 1.0m,

    Taxi {Fares: 0}  => 3.50m + 1.00m,
    Taxi {Fares: 1}  => 3.50m,
    Taxi {Fares: 2}  => 3.50m - 0.50m,
    Taxi             => 3.50m - 1.00m,

    Bus b when ((double)b.Riders / (double)b.Capacity) < 0.50 => 5.00m +
2.00m,
    Bus b when ((double)b.Riders / (double)b.Capacity) > 0.90 => 5.00m -
1.00m,
    Bus => 5.00m,

    DeliveryTruck t when (t.GrossWeightClass > 5000) => 10.00m + 5.00m,
    DeliveryTruck t when (t.GrossWeightClass < 3000) => 10.00m - 2.00m,
    DeliveryTruck => 10.00m,

    { }     => throw new ArgumentException(message: "Not a known vehicle
type", paramName: nameof(vehicle)),
    null    => throw new ArgumentNullException(nameof(vehicle))
};
```

Many of these switch arms are examples of **recursive patterns**. For example, `Car { Passengers: 1}` shows a constant pattern inside a property pattern.

You can make this code less repetitive by using nested switches. The `Car` and `Taxi` both have four different arms in the preceding examples. In both cases, you can create a declaration pattern that feeds into a constant pattern. This technique is shown in the following code:

C#

```
public decimal CalculateToll(object vehicle) =>
    vehicle switch
    {
        Car c => c.Passengers switch
        {
            0 => 2.00m + 0.5m,
            1 => 2.0m,
```

```
            2 => 2.0m - 0.5m,
            _ => 2.00m - 1.0m
        },

        Taxi t => t.Fares switch
        {
            0 => 3.50m + 1.00m,
            1 => 3.50m,
            2 => 3.50m - 0.50m,
            _ => 3.50m - 1.00m
        },

        Bus b when ((double)b.Riders / (double)b.Capacity) < 0.50 => 5.00m +
2.00m,
        Bus b when ((double)b.Riders / (double)b.Capacity) > 0.90 => 5.00m -
1.00m,
        Bus b => 5.00m,

        DeliveryTruck t when (t.GrossWeightClass > 5000) => 10.00m + 5.00m,
        DeliveryTruck t when (t.GrossWeightClass < 3000) => 10.00m - 2.00m,
        DeliveryTruck t => 10.00m,

        { }  => throw new ArgumentException(message: "Not a known vehicle
type", paramName: nameof(vehicle)),
        null => throw new ArgumentNullException(nameof(vehicle))
    };
```

In the preceding sample, using a recursive expression means you don't repeat the `Car` and `Taxi` arms containing child arms that test the property value. This technique isn't used for the `Bus` and `DeliveryTruck` arms because those arms are testing ranges for the property, not discrete values.

# Add peak pricing

For the final feature, the toll authority wants to add time sensitive peak pricing. During the morning and evening rush hours, the tolls are doubled. That rule only affects traffic in one direction: inbound to the city in the morning, and outbound in the evening rush hour. During other times during the workday, tolls increase by 50%. Late night and early morning, tolls are reduced by 25%. During the weekend, it's the normal rate, regardless of the time. You could use a series of `if` and `else` statements to express this using the following code:

C#

```
public decimal PeakTimePremiumIfElse(DateTime timeOfToll, bool inbound)
{
    if ((timeOfToll.DayOfWeek == DayOfWeek.Saturday) ||
        (timeOfToll.DayOfWeek == DayOfWeek.Sunday))
    {
        return 1.0m;
    }
    else
    {
        int hour = timeOfToll.Hour;
        if (hour < 6)
        {
            return 0.75m;
        }
        else if (hour < 10)
        {
            if (inbound)
            {
                return 2.0m;
            }
            else
            {
                return 1.0m;
            }
        }
        else if (hour < 16)
        {
            return 1.5m;
        }
        else if (hour < 20)
        {
            if (inbound)
            {
                return 1.0m;
            }
            else
            {
                return 2.0m;
            }
        }
        else // Overnight
        {
            return 0.75m;
        }
    }
}
```

The preceding code does work correctly, but isn't readable. You have to chain through

all the input cases and the nested `if` statements to reason about the code. Instead, you'll use pattern matching for this feature, but you'll integrate it with other techniques. You could build a single pattern match expression that would account for all the combinations of direction, day of the week, and time. The result would be a complicated expression. It would be hard to read and difficult to understand. That makes it hard to ensure correctness. Instead, combine those methods to build a tuple of values that concisely describes all those states. Then use pattern matching to calculate a multiplier for the toll. The tuple contains three discrete conditions:

- The day is either a weekday or a weekend.
- The band of time when the toll is collected.
- The direction is into the city or out of the city

The following table shows the combinations of input values and the peak pricing multiplier:

| Day | Time | Direction | Premium |
|---|---|---|---|
| Weekday | morning rush | inbound | x 2.00 |
| Weekday | morning rush | outbound | x 1.00 |
| Weekday | daytime | inbound | x 1.50 |
| Weekday | daytime | outbound | x 1.50 |
| Weekday | evening rush | inbound | x 1.00 |
| Weekday | evening rush | outbound | x 2.00 |
| Weekday | overnight | inbound | x 0.75 |
| Weekday | overnight | outbound | x 0.75 |
| Weekend | morning rush | inbound | x 1.00 |
| Weekend | morning rush | outbound | x 1.00 |
| Weekend | daytime | inbound | x 1.00 |
| Weekend | daytime | outbound | x 1.00 |
| Weekend | evening rush | inbound | x 1.00 |
| Weekend | evening rush | outbound | x 1.00 |

| Day | Time | Direction | Premium |
|---|---|---|---|
| Weekend | overnight | inbound | x 1.00 |
| Weekend | overnight | outbound | x 1.00 |

There are 16 different combinations of the three variables. By combining some of the conditions, you'll simplify the final switch expression.

The system that collects the tolls uses a DateTime structure for the time when the toll was collected. Build member methods that create the variables from the preceding table. The following function uses a pattern matching switch expression to express whether a DateTime represents a weekend or a weekday:

C#

```
private static bool IsWeekDay(DateTime timeOfToll) =>
    timeOfToll.DayOfWeek switch
    {
        DayOfWeek.Monday    => true,
        DayOfWeek.Tuesday   => true,
        DayOfWeek.Wednesday => true,
        DayOfWeek.Thursday  => true,
        DayOfWeek.Friday    => true,
        DayOfWeek.Saturday  => false,
        DayOfWeek.Sunday    => false
    };
```

That method is correct, but it's repetitious. You can simplify it, as shown in the following code:

C#

```
private static bool IsWeekDay(DateTime timeOfToll) =>
    timeOfToll.DayOfWeek switch
    {
        DayOfWeek.Saturday => false,
        DayOfWeek.Sunday => false,
        _ => true
    };
```

Next, add a similar function to categorize the time into the blocks:

C#

```
private enum TimeBand
{
    MorningRush,
    Daytime,
    EveningRush,
    Overnight
}

private static TimeBand GetTimeBand(DateTime timeOfToll) =>
    timeOfToll.Hour switch
    {
        < 6 or > 19 => TimeBand.Overnight,
        < 10 => TimeBand.MorningRush,
        < 16 => TimeBand.Daytime,
        _ => TimeBand.EveningRush,
    };
```

You add a private `enum` to convert each range of time to a discrete value. Then, the `GetTimeBand` method uses relational patterns, and conjunctive or patterns, both added in C# 9.0. A relational pattern lets you test a numeric value using `<`, `>`, `<=`, or `>=`. The `or` pattern tests if an expression matches one or more patterns. You can also use an `and` pattern to ensure that an expression matches two distinct patterns, and a `not` pattern to test that an expression doesn't match a pattern.

After you create those methods, you can use another `switch` expression with the **tuple pattern** to calculate the pricing premium. You could build a `switch` expression with all 16 arms:

C#

```
public decimal PeakTimePremiumFull(DateTime timeOfToll, bool inbound) =>
    (IsWeekDay(timeOfToll), GetTimeBand(timeOfToll), inbound) switch
    {
        (true, TimeBand.MorningRush, true) => 2.00m,
        (true, TimeBand.MorningRush, false) => 1.00m,
        (true, TimeBand.Daytime, true) => 1.50m,
        (true, TimeBand.Daytime, false) => 1.50m,
        (true, TimeBand.EveningRush, true) => 1.00m,
        (true, TimeBand.EveningRush, false) => 2.00m,
        (true, TimeBand.Overnight, true) => 0.75m,
        (true, TimeBand.Overnight, false) => 0.75m,
        (false, TimeBand.MorningRush, true) => 1.00m,
        (false, TimeBand.MorningRush, false) => 1.00m,
        (false, TimeBand.Daytime, true) => 1.00m,
```

```
        (false, TimeBand.Daytime, false) => 1.00m,
        (false, TimeBand.EveningRush, true) => 1.00m,
        (false, TimeBand.EveningRush, false) => 1.00m,
        (false, TimeBand.Overnight, true) => 1.00m,
        (false, TimeBand.Overnight, false) => 1.00m,
    };
```

The above code works, but it can be simplified. All eight combinations for the weekend have the same toll. You can replace all eight with the following line:

C#

```
(false, _, _) => 1.0m,
```

Both inbound and outbound traffic have the same multiplier during the weekday daytime and overnight hours. Those four switch arms can be replaced with the following two lines:

C#

```
(true, TimeBand.Overnight, _) => 0.75m,
(true, TimeBand.Daytime, _)   => 1.5m,
```

The code should look like the following code after those two changes:

C#

```
public decimal PeakTimePremium(DateTime timeOfToll, bool inbound) =>
    (IsWeekDay(timeOfToll), GetTimeBand(timeOfToll), inbound) switch
    {
        (true, TimeBand.MorningRush, true)  => 2.00m,
        (true, TimeBand.MorningRush, false) => 1.00m,
        (true, TimeBand.Daytime,     _)     => 1.50m,
        (true, TimeBand.EveningRush, true)  => 1.00m,
        (true, TimeBand.EveningRush, false) => 2.00m,
        (true, TimeBand.Overnight,   _)     => 0.75m,
        (false, _,                   _)     => 1.00m,
    };
```

Finally, you can remove the two rush hour times that pay the regular price. Once you remove those arms, you can replace the `false` with a discard (`_`) in the final switch arm. You'll have the following finished method:

C#

```
public decimal PeakTimePremium(DateTime timeOfToll, bool inbound) =>
    (IsWeekDay(timeOfToll), GetTimeBand(timeOfToll), inbound) switch
    {
        (true, TimeBand.Overnight, _) => 0.75m,
        (true, TimeBand.Daytime, _) => 1.5m,
        (true, TimeBand.MorningRush, true) => 2.0m,
        (true, TimeBand.EveningRush, false) => 2.0m,
        _ => 1.0m,
    };
```

This example highlights one of the advantages of pattern matching: the pattern branches are evaluated in order. If you rearrange them so that an earlier branch handles one of your later cases, the compiler warns you about the unreachable code. Those language rules made it easier to do the preceding simplifications with confidence that the code didn't change.

Pattern matching makes some types of code more readable and offers an alternative to object-oriented techniques when you can't add code to your classes. The cloud is causing data and functionality to live apart. The *shape* of the data and the *operations* on it aren't necessarily described together. In this tutorial, you consumed existing data in entirely different ways from its original function. Pattern matching gave you the ability to write functionality that overrode those types, even though you couldn't extend them.

# Next steps

You can download the finished code from the dotnet/samples GitHub repository. Explore patterns on your own and add this technique into your regular coding activities. Learning these techniques gives you another way to approach problems and create new functionality.

# See also

- Patterns
- switch expression