**Starting point**

Open Visual Studio and create a new blank solution. We’ll simulate a simple shipping cost calculation application where the calculation will depend on the type of the carrier: FedEx, UPS, Schenker. Add a Class Library called Domain with the following three classes:

Address.cs:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | public class Address      {          public string ContactName { get; set; }          public string AddressLine1 { get; set; }          public string AddressLine2 { get; set; }          public string AddressLine3 { get; set; }          public string City { get; set; }          public string Region { get; set; }          public string Country { get; set; }            public string PostalCode { get; set; }      } |

ShippingOptions enumeration

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | public enum ShippingOptions      {          UPS = 100,          FedEx = 200,          Schenker = 300,      } |

Order.cs:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | public class Order      {          public ShippingOptions ShippingMethod { get; set; }          public Address Destination { get; set; }          public Address Origin { get; set; }      } |

The domain objects shouldn’t be terribly difficult to understand. The shipping cost calculation is definitely a domain logic so we’ll include the following domain service in the Domain layer:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37 | public class ShippingCostCalculatorService      {          public double CalculateShippingCost(Order order)          {              switch (order.ShippingMethod)              {                  case ShippingOptions.FedEx:                      return CalculateForFedEx(order);                    case ShippingOptions.UPS:                      return CalculateForUPS(order);                    case ShippingOptions.Schenker:                      return CalculateForSchenker(order);                    default:                      throw new Exception("Unknown carrier");                }            }            double CalculateForSchenker(Order order)          {              return 3.00d;          }            double CalculateForUPS(Order order)          {              return 4.25d;          }            double CalculateForFedEx(Order order)          {              return 5.00d;          }      } |

If you are not familiar with patterns and practices then this implementation might look perfectly normal to you. We check the type of the shipping method in an enumeration and then call on a method to calculate the cost accordingly.

**What’s wrong with this code?**

It is perfectly reasonable that we may introduce a new carrier in the future, say PerfectHaulage. If we pass an order with this shipping method to the CalculateShippingCost method then we’ll get an exception. We’d have to manually extend the switch statement to account for the new shipment type. In case of a new carrier we’d have to come back to this domain service and modify it accordingly. That breaks the Open/Closed principle of SOLID: a class is open for extensions but closed for modifications. In addition, if there’s a change in the implementation of one of the calculation algorithms then again we’d have to come back to this method and modify it. That’s generally not a good practice: if you make a change to one of your classes, then you should not have to go an modify other classes and public methods just to accommodate that change.

If a change in one part of the project necessitates changes in another part of the project then chances are that we’re facing some serious coupling issues between two or more classes. If you repeatedly have to revisit other parts of your software after changing some class then you probably have a design issue.

Also, the methods that calculate the costs are of course ridiculously simple in this demo – in reality there may well be calls to other services, the weight of the package may be checked etc., so the ShippingCostCalculatorService class may grow very large and difficult to maintain. The calculator class becomes bloated with logic belonging to UPS, FedEx, DHL etc, violating the Single Responsibility Principle. The service class is trying to take care of too much.

There are problems related to the design of the domain. We could argue that it is not the responsibility of the Order domain to know which shipping method a customer prefers. This is up to the domain experts to decide, but this is certainly a possible implementation problem.