Common Patterns and Antipatterns

Common C# Patterns and Antipatterns

Common approaches to solving similar problems are referred to as *design patterns*. Common approaches to solving similar problems that often end up causing more problems than they solve are called *antipatterns* (or *anti-patterns*). As you continue learning C#, there are a few of each to keep in mind.

Note: Learn more about design patterns and antipatterns.

In this lesson, you're going to learn about the *adapter*, *factory*, *repository*, and *strategy*design patterns. You'll also learn about two closely-related antipatterns: *Singleton* and *Static Cling*. As you complete this lesson, keep in mind the new is glue tip you learned about earlier in this tutorial.

The Adapter Design Pattern

The goal of the Adapter design pattern is to convert one interface to another. Frequently this is to allow multiple different systems or objects to interact with one another. You may have code that needs to work simultaneously with many different systems, or needs to be able to switch between them easily (either at runtime or with a simple change in configuration).

As an example, imagine you're working on a system that can integrate with multiple payment providers. Based on the preferred provider, the system will accept payments. However, each provider uses its own API to accept payments. Let's look at the two providers you're going to start with, *Stwipe* and *PaySal*:

```
public class StwipeProvider
{
    public StwipeProvider(string merchantKey)
    {}

    // returns false if payment is rejected
    public bool Pay(string cardNumber, string expiration, decimal amount)
    {}
}

public class PaySalProvider
{
    // throws exception if payment is rejected
    public void ProcessPayment(string merchantId, CreditCardDetails cardInfo, decimal amount
    {}
}
```

You expect that you're going to support additional payment providers in the future, and it's also likely that at some point these providers may change their APIs. Thus, you want to shield your code from potential breaking changes in the future. The way to achieve this is to create your own Adapter interface that you will work with, and write your own Adapter implementations for each provider. These should be the only classes in your application that reference the provider-specific code - the rest of your application should work only with your adapters.

When creating your adapter interface, you're free to make whatever design decisions will make it easiest for your application to work with it. Of course, the closer you make it to at least one of the interfaces it will be adapting, the less work you'll need to do when writing (at least one of) the adapter implementations. Note that the `PaySalProvider` interface accepts a `CreditCardDetails` type. This is a custom type defined specifically in the `PaySalProvider` package. Your adapter will need to avoid using any provider-specific types in its interface.

```
public interface IPaymentProcessorAdapter
{
    // returns false if payment is rejected
    bool ProcessPayment(string merchantId, string cardNumber, string expiration,
                        decimal amount);
}

public StwipeAdapter : IPaymentProcessorAdapter
{
    public bool ProcessPayment(string merchantId, string cardNumber, string expiration,
                        decimal amount)
    {
        var provider = new StwipeProvider(merchantId);
        return provider.Pay(cardNumber, expiration, amount);
    }
}

public PaySalAdapter : IPaymentProcessorAdapter
{
    public bool ProcessPayment(string merchantId,
        string cardNumber, string expiration,
        decimal amount)
    {
        var provider = new PaySalProvider();
        try
        {
            var cardInfo = new CreditCardDetails(cardNumber, expiration);
            provider.ProcessPayment(merchantId, cardInfo, amount);
            return true;
        }
        catch
        {
            return false;
        }
    }
}
```

With this design in place, you could accept an IPaymentProcessorAdapter anywhere you needed to process a payment. You could set it as a property on a user or storefront object to specify its payment preference. And you could add many additional adapters as your support for other payment providers continued to grow, all without any of the rest of your application having to deal with these updates directly.

The Factory Design Pattern

The factory design pattern is actually a combination of a few different patterns, but with the same intent: to simplify construction of object instances and encapsulate decisions about which specific type to instantiate. For building on the example above, let's say the payment providers are used by different stores that all run on your platform. Each store must choose a particular payment provider they're going to use from the ones the platform supports. This is stored as part of the `Store` 's properties:

```
public class Store
{
    public string Name { get; set; }
    public string PaymentProvider { get; set; }
    public string MerchantId { get; set; }
}
```

When someone buys something from the store, there's a method somewhere that is responsible for processing the payment. It might belong to the `Store` class, or it might live somewhere else, but regardless it's going to need to create the appropriate adapter class in order to make the payment. The method might look something like this:

```
public void ProcessCard(string cardNumber, string expiration, decimal amount)
{
    IPaymentProcessorAdapter adapter = null;
    if (PaymentProvider = "Stwipe")
    {
        adapter = new StwipeAdapter();
    }
    else if (PaymentProvider = "PaySal")
    {
        adapter = new PaySalAdapter();
    }
    else
    {
        throw new InvalidPaymentProviderException(PaymentProvider);
    }
    adapter.Pay(MerchantId, cardNumber, expiration, amount);
}
```

As you can see, the bulk of this method is concerned with determining which specific adapter to instantiate, not the logic of processing the card. This construction logic can be moved into its own *factory method*:

```
public void ProcessCard(string cardNumber, string expiration, decimal amount)
{
    IPaymentProcessorAdapter adapter = GetPaymentAdapter();

    adapter.Pay(MerchantId, cardNumber, expiration, amount);
}

private IPaymentProcessorAdapter GetPaymentAdapter()
{
    if (PaymentProvider = "Stwipe")
    {
        adapter = new StwipeAdapter();
    }
    else if (PaymentProvider = "PaySal")
    {
        adapter = new PaySalAdapter();
    }
    else
    {
        throw new InvalidPaymentProviderException(PaymentProvider);
    }
}
```

You can take this further, and move the logic into its own type, so that the class doing the payment processing doesn't have the added responsibility of determining how to create the adapter. Typically this type will use the "Factory" suffix in its name:

```
public interface IPaymentProcessorAdapterFactory
{
    IPaymentProcessorAdapter Create(string providerName);
}

public PaymentProcessorAdapterFactory : IPaymentProcessorAdapterFactory
{
    public IPaymentProcessorAdapter Create(string providerName)
    {
        if (PaymentProvider = "Stwipe")
        {
            adapter = new StwipeAdapter();
        }
        else if (PaymentProvider = "PaySal")
        {
            adapter = new PaySalAdapter();
        }
        else
        {
            throw new InvalidPaymentProviderException(PaymentProvider);
        }
    }
}
```

Now the responsibility for creating the correct adapter (which could require much more complexity than is shown here, and which might have far more than two valid options) has been moved into its own class. As it stands now, the `ProcessCard` method would probably need to directly instantiate the factory class in order to use it, but you'll see below how the *Strategy* pattern can address this.

The Repository Design Pattern

Of course, when an order is placed and payment succeeds, the order needs to be stored somewhere. The `Store` itself could include logic for connecting to a database and executing commands against it to perform this logic, but, once more, that's an additional responsibility the store shouldn't take on for itself. Rather, some other class should have the responsibility of persistence (of `Orders` in this case). There is a design pattern for encapsulating persistence operations behind a class with a collection-like interface, and it is called the *Repository pattern*.

The goal of this pattern is to make working with external persistence mechanisms, like databases, as simple for the application code as working with a built-in collection would be. Thus, when you define an interface for a repository, it will typically accept parameters like *Add* and *Remove* and *Get*, but usually this similarity stops short of having the repository type implement *ICollection* or *IEnumerable* directly.

Connecting to different data stores is outside the scope of this tutorial, but the details aren't important. Consider the following block of code:

```
public void CompleteOrder()
{
    // verify order

    // process card

    // create order object

    // connect to database
    // convert order object into database statement(s)
    // execute database commands

    // send customer confirmation
}
```

You don't want the low-level details of connecting to a database to be in the middle of a high level business method on completing an order. At the very least, those details should be extracted into their own method. However, there are likely to be many similar such methods, all concerned with the specifics of data access. It's far more cohesive to put these methods on classes whose specific responsibility is persistence. These are called repositories.

```
public interface IOrderRepository
{
    void Add(Order order);
}
public class DbOrderRepository : IOrderRepository
{
```

```
    public void Add(Order order);
}
```

Now, the `CompleteOrder` method can simply refer to an instance of `IOrderRepository`:

```
public void CompleteOrder()
{
    // other logic omitted
    orderRepository.Add(order);
}
```

Or course, in addition to adding records, your applications will probably need to read them, update them, and delete them. These operations (Create, Read, Update, Delete) are often referred to by the acronym *CRUD*. Repositories are where you should typically implement your CRUD logic in your applications. Add additional methods to the initial interface as your requirements demand them. A more complete `IOrderRepository` interface might look like this:

```
public interface IOrderRepository
{
    Order GetById(int id);
    List<Order> List();
    void Add(Order order);
    void Update (Order order);
    void Delete (Order order);
}
```

With such an abstraction in place, you can perform most data access operations from your business-level code without coupling it directly to any particular persistence implementation. This also helps your business classes remain persistence ignorant(again, keeping them from being coupled to a particular persistence implementation).

Once you've defined an interface to encapsulate your persistence operations, you can write the code that works with this interface. However, before you can run it, you need to write an implementation of the interface, and use this implementation from your code. If you simply instantiate it, you're still gluing your code to that implementation. A better approach that results in a more modular design is to take in the interface as a parameter, as you'll learn in the next section.

The Strategy Design Pattern

The *Strategy* design pattern allows an object to have some of the details of its behavior encapsulated in another type, which is then provided to it as a parameter. This pattern is closely related to dependency injection, which refers to the technique of passing dependencies (or *injecting* them) into classes, usually through their constructor.

Returning to the example from the previous section, how does the `CompleteOrder` method get an instance of `IOrderRepository`? One approach that will cause coupling problems is to simply instantiate an instance directly in the method:

```
public void CompleteOrder()
{
    // other logic omitted
    IOrderRepository orderRepository = new EntityFrameworkOrderRepository();
    orderRepository.Add(order);
}
```

This approach is inflexible, and violates the Open-Closed Principle, because the only way to change the persistence behavior in the future is to modify this code. Following the strategy pattern, you would identify the `IOrderRepository` as a *strategy* for persistence operations. You would then pass in the implementation of this strategy as a parameter to the class (or, less commonly, as a property or method parameter). The updated design using this pattern would be:

```
public class Order
{
    private readonly IOrderRepository _orderRepository;
    public Order(IOrderRepository orderRepository)
    {
        _orderRepository = orderRepository;
    }
    public void CompleteOrder()
    {
        // other logic omitted
        _orderRepository.Add(order);
    }
}
```

This class now follows the Explicit Dependencies Principle, because it clearly states in its constructor what its needed collaborators are. It also better follows SRP (discussed in the encapsulation lesson), because it is no longer responsible for choosing the specific implementations it will collaborate with. That decision can be made elsewhere, perhaps in a class whose sole responsibility is constructing the collaborators that the program will use.

The Singleton Antipattern

The *Singleton* design pattern is commonly considered an antipattern. For some types, having more than one instance of the type within an application could result in adverse effects. The pattern addresses this by having the type itself take responsibility for ensuring this behavior. There's nothing wrong with having different lifetime behaviors for different types, but it violates SRP to add lifetime management responsibility to a class that already does something else. Further, this design tends to introduce tight coupling within the application between the Singleton types and those that refer to them.

A typical example of the Singleton pattern:

```
public sealed class OrderProcessor
{
    // constructor is private; class cannot be instantiated externally
    private OrderProcessor() {}
```

```
    private static OrderProcessor _instance; // a static instance; only one exists within th
    public OrderProcessor Instance
    {
        get
        {
            // bad code - do not do this
            if (_instance == null)
            {
                _instance = new OrderProcessor();
            }
            return _instance;
        }
    }
}
```

In addition to the design problems this pattern can introduce, the naive implementation shown above can have problems in multi-threaded applications. Learn more about implementing the Singleton pattern in C#, to see some different approaches to this pattern that can address some of its deficiencies (but not the coupling it creates).

> **Note** {.note}
> To avoid tightly coupling your code to static implementations, favor the use of dependency injection and the Strategy design pattern. Then, your code that is responsible for instantiating the types your application uses at runtime can determine the objects' lifetimes. For some, a new instance may be used by every type that requests one. For others, the same instance may be used for the life of the application - the same behavior the Singleton pattern achieves, but without its negative consequences.

The Static Cling Antipattern

The *Static Cling* antipattern gets its name from the *static* keyword in C#, which makes code constructs global within an application. Similar to the Singleton, references to these global types, properties, and methods result in tight coupling, hence the use of the term *static cling* referring to how this use makes parts of the application stick together.

> **Note** {.note}
> As with the Singleton, the preferred approach is to have types be explicit about their dependencies and to inject them in using the Strategy pattern. Existing code that leverages static resources can be wrapped in Adapter implementations to achieve this same result.