**http://dotnetcodr.com/architecture-and-patterns/**

**SOLID design principles in .NET: the Single Responsibility Principle**

August 12, 2013[11 Comments](http://dotnetcodr.com/2013/08/12/solid-design-principles-in-net-the-single-responsibility-principle/#comments)

The SOLID design principles are a collection of best practices for object oriented software design. The abbreviation comes from the first letter of each of the following 5 constituents:

* Single responsibility principle (SRP)
* Open-Closed principle (OCP)
* Liskov substitution principle (LSP)
* Interface segregation principle (ISP)
* Dependency inversion principle (DIP)

Each of these terms are meant to make your code base easier to understand and maintain. They also ensure that your code does not become a mess with a large degree of interdependency that nobody wants to debug and extend. Of course you can write functioning software without adhering to these guidelines but they are a good investment in the future development of your product especially as far as maintainability and extensibility are concerned. Also, by following these points your code will become more object oriented instead of employing a more procedural style of coding with a lot of magic strings and enumerations and other primitives.

However, the principles do not replace the need for maintaining and refactoring your code so that it doesn’t get stale. They are a good set of tools to make your future work with your code easier. We will look at each of these in this series.

You can view these principles as guidelines. You should write code with these guidelines in mind and should aim to get as far as possible to reach each of them. You won’t always succeed of course, but even a bit of SOLID is more than the total lack of it.

**SRP introduction**

The Single Responsibility Principle states that every object should only have one reason to change and a single focus of responsibility. In other words every object should perform one thing only. You can apply this idea at different levels of your software: a method should only carry out one action; a domain object should only represent one domain within your business; the presentation layer should only be responsible for presenting your data; etc. This principle aims to achieve the following goals:

* Short and concise objects: avoid the problem of a monolithic class design that is the software equivalent of a Swiss army knife
* Testability: if a method carries out multiple tasks then it’s difficult to write a test for it
* Readability: reading short and concise code is certainly easier than finding your way through some spaghetti code
* Easier maintenance

A responsibility of a class usually represents a feature or a domain in your application. If you assign many responsibilities to a class or bloat your domain object then there’s a greater chance that you’ll need to change that class later. These responsibilities will be coupled together in the class making each individual responsibility more difficult to change without introducing errors in another. We can also call a responsibility a “reason to change”.

SRP is strongly related to what is called **Separation of Concerns (SoC)**. SoC means dissecting a piece of software into distinct features that encapsulate unique behaviour and data that can be used by other classes. Here the term ‘concern’ represents a feature or behaviour of a class. Separating a programme into small and discrete ‘ingredients’ significantly increases code reuse, maintenance and testability.

Other related terms include the following:

* **Cohesion**: how strongly related and focused the various responsibilities of a module are
* **Coupling**: the degree to which each programme module relies on each one of the other modules

In a good software design we are striving for a high level of cohesion and a low level of coupling. A high level of coupling, also called tight coupling, usually means a lot of concrete dependency among the various elements of your software. This leads to a situation where changing the design of one class leads to the need of changing other classes that are dependent on the class you’ve just changed. Also, with tight coupling changing the design of one class can introduce errors in the dependent classes.

One last related technique is Test Driven Design or Test Driven Development (TDD). If you apply the **test first approach** of TDD and write your tests carefully then it will help you fulfil SRP, or at least it is a good way to ensure that you’re not too far from SRP. If you don’t know what TDD is then you can read about it [here](http://dotnetcodr.com/2013/03/25/test-driven-development-in-net-part-1-the-absolute-basics-of-red-green-refactor/).

**Demo**

In the demo we’ll simulate an e-commerce application. We’ll first deliberately introduce a bloated Order object with a lot of responsibilities. We’ll then refactor the code to get closer to SRP.

Open Visual Studio and create a new console application. Insert a new folder called Model and insert a couple of basic models into it:

|  |  |
| --- | --- |
| 1  2  3  4  5 | public class OrderItem  {  public string Identifier { get; set; }  public int Quantity { get; set; }  } |
| 1  2  3  4  5  6 | public class ShoppingCart  {  public decimal TotalAmount { get; set; }  public IEnumerable<OrderItem> Items { get; set; }  public string CustomerEmail { get; set; }  } |

|  |  |
| --- | --- |
| 1  2  3  4  5 | public enum PaymentMethod  {  CreditCard  , Cheque  } |
| 1  2  3  4  5  6  7 | public class PaymentDetails  {  public PaymentMethod PaymentMethod { get; set; }  public string CreditCardNumber { get; set; }  public DateTime ExpiryDate { get; set; }  public string CardholderName { get; set; }  } |

This is all pretty simple up this point I believe. Now comes the most important domain object, Order, which has quite many areas of responsibility:

|  |  |
| --- | --- |
| 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  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87 | public class Order  {  public void Checkout(ShoppingCart shoppingCart, PaymentDetails paymentDetails, bool notifyCustomer)  {  if (paymentDetails.PaymentMethod == PaymentMethod.CreditCard)  {  ChargeCard(paymentDetails, shoppingCart);  }  ReserveInventory(shoppingCart);  if (notifyCustomer)  {  NotifyCustomer(shoppingCart);  }  }  public void NotifyCustomer(ShoppingCart cart)  {  string customerEmail = cart.CustomerEmail;  if (!String.IsNullOrEmpty(customerEmail))  {  try  {  //construct the email message and send it, implementation ignored  }  catch (Exception ex)  {  //log the emailing error, implementation ignored  }  }  }  public void ReserveInventory(ShoppingCart cart)  {  foreach (OrderItem item in cart.Items)  {  try  {  InventoryService inventoryService = new InventoryService();  inventoryService.Reserve(item.Identifier, item.Quantity);  }  catch (InsufficientInventoryException ex)  {  throw new OrderException("Insufficient inventory for item " + item.Sku, ex);  }  catch (Exception ex)  {  throw new OrderException("Problem reserving inventory", ex);  }  }  }  public void ChargeCard(PaymentDetails paymentDetails, ShoppingCart cart)  {  PaymentService paymentService = new PaymentService();  try  {  paymentService.Credentials = "Credentials";  paymentService.CardNumber = paymentDetails.CreditCardNumber;  paymentService.ExpiryDate = paymentDetails.ExpiryDate;  paymentService.NameOnCard = paymentDetails.CardholderName;  paymentService.AmountToCharge = cart.TotalAmount;  paymentService.Charge();  }  catch (AccountBalanceMismatchException ex)  {  throw new OrderException("The card gateway rejected the card based on the address provided.", ex);  }  catch (Exception ex)  {  throw new OrderException("There was a problem with your card.", ex);  }  }  }  public class OrderException : Exception  {  public OrderException(string message, Exception innerException)  : base(message, innerException)  {  }  } |

The Order class won’t compile yet, so add a new folder called Services with the following objects representing the Inventory and Payment services:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | public class InventoryService  {  public void Reserve(string identifier, int quantity)  {  throw new InsufficientInventoryException();  }  }  public class InsufficientInventoryException : Exception  {  } |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | public class PaymentService  {  public string CardNumber { get; set; }  public string Credentials { get; set; }  public DateTime ExpiryDate { get; set; }  public string NameOnCard { get; set; }  public decimal AmountToCharge { get; set; }  public void Charge()  {  throw new AccountBalanceMismatchException();  }  }  public class AccountBalanceMismatchException : Exception  {  } |

These are two very simple services with no real implementation that only throw exceptions.

Looking at the Order class we see that it performs a lot of stuff: checking out after the customer has placed an order, sending emails, logging exceptions, charging the credit card etc. Probably the most important method here is Checkout which calls upon the other methods in the class.

What is the problem with this design? After all it works well, customers can place orders, they get notified etc.

I think first and foremost the greatest flaw is a conceptual one actually. What has the Order domain object got to do with sending emails? What does it have to do with checking the inventory, logging exceptions or charging the credit card? These are all concepts that simply do not belong in an Order domain.

Imagine that the Order object can be used by different platforms: an e-commerce website with credit card payments or a physical shop where you pick your own goods from the shelf and pay by cash. Which leads to several other issues as well:

* Cheque payments don’t need card processing: cards are only charged in the Checkout method if the customer is paying by card – in any other case we should not involve the idea of card processing at all
* Inventory reservations should be carried out by a separate service in case we’re buying in a physical shop
* The customer will probably only need an email notification if they use the web platform of the business – otherwise the customer won’t even provide an email address. After all, why would you want to be notified by email if you buy the goods in person in a shop?

The problem here is that no matter what platform consumes the Order object it will need to know about the concepts of inventory management, credit card processing and emails. So any change in these concepts will affect not only the Order object but all others that depend on it.

Let’s refactor to a better design. The key is to regroup the responsibilities of the Checkout method into smaller units. Add a new folder called SRP to the project so that you’ll have access to the objects before and after the refactoring.

We know that we can process several types of Order: an online order, a cash order, a cheque order and possibly other types of Order that we haven’t thought of. This calls for an abstract Order object:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | public abstract class Order  {  private readonly ShoppingCart \_shoppingCart;  public Order(ShoppingCart shoppingCart)  {  \_shoppingCart = shoppingCart;  }  public ShoppingCart ShoppingCart  {  get  {  return \_shoppingCart;  }  }  public virtual void Checkout()  {  //add common functionality to all Checkout operations  }  } |

We’ll separate out the responsibilities of the original Checkout method into interfaces:

|  |  |
| --- | --- |
| 1  2  3  4 | public interface IReservationService  {  void ReserveInventory(IEnumerable<OrderItem> items);  } |
| 1  2  3  4 | public interface IPaymentService  {  void ProcessCreditCard(PaymentDetails paymentDetails, decimal moneyAmount);  } |

|  |  |
| --- | --- |
| 1  2  3  4 | public interface INotificationService  {  void NotifyCustomerOrderCreated(ShoppingCart cart);  } |

So we separated out the inventory management, customer notification and payment services into their respective interfaces. We can now create some concrete Order objects. The simplest case is when you go to a shop, place your goods into a real shopping cart and pay at the cashier. There’s no credit card process and no email notification. Also, the inventory has probably been reduced when the goods were placed on the shelf, there’s no need to reduce the inventory further when the actual purchase happens:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | class CashOrder : Order  {  public CashOrder(ShoppingCart shoppingCart)  : base(shoppingCart)  { }  } |

That’s all for the cash order which represents an immediate purchase in a shop where the customer pays by cash. You can of course pay by credit card in a shop so let’s create another order type:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | public class CreditCardOrder : Order  {  private readonly PaymentDetails \_paymentDetails;  private readonly IPaymentService \_paymentService;  public CreditCardOrder(ShoppingCart shoppingCart  , PaymentDetails paymentDetails, IPaymentService paymentService) : base(shoppingCart)  {  \_paymentDetails = paymentDetails;  \_paymentService = paymentService;  }  public override void Checkout()  {  \_paymentService.ProcessCreditCard(\_paymentDetails, ShoppingCart.TotalAmount);  base.Checkout();  }  } |

The credit card payment must be processed hence we’ll need a Payment service to take care of that. We call upon its ProcessCreditCard method in the overridden Checkout method. Here the consumer platform can provide some concrete implementation of the IPaymentService interface, it doesn’t matter to the Order object.

Lastly we can have an online order with inventory management, payment service and email notifications:

|  |  |
| --- | --- |
| 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 | public class OnlineOrder : Order  {  private readonly INotificationService \_notificationService;  private readonly PaymentDetails \_paymentDetails;  private readonly IPaymentService \_paymentService;  private readonly IReservationService \_reservationService;  public OnlineOrder(ShoppingCart shoppingCart,  PaymentDetails paymentDetails, INotificationService notificationService  , IPaymentService paymentService, IReservationService reservationService)  : base(shoppingCart)  {  \_paymentDetails = paymentDetails;  \_paymentService = paymentService;  \_reservationService = reservationService;  \_notificationService = notificationService;  }  public override void Checkout()  {  \_paymentService.ProcessCreditCard(\_paymentDetails, ShoppingCart.TotalAmount);  \_reservationService.ReserveInventory(ShoppingCart.Items);  \_notificationService.NotifyCustomerOrderCreated(ShoppingCart);  base.Checkout();  }  } |

The consumer application will provide concrete implementations for the notification, inventory management and payment services. The OnlineOrder object will not care what those implementations look like and will not be affected at all if you make a change in those implementations or send in a different concrete implementation. As you can see these are the responsibilities that are likely to change over time. However, the Order object and its concrete implementations won’t care any more.

Furthermore, a web platform will only concern itself with online orders now and not with point-of-sale ones such as CreditOrder and CashOrder. The platform that a cashier uses in the shop will probably use CashOrder and CreditOrder objects depending on the payment method of the customer. The web and point-of-sale platforms will no longer be affected by changes made to the inventory management, email notification and payment processing logic.

Also, note that we separated out the responsibilities into individual smaller interfaces and not a single large one with all responsibilities. This follows the letter ‘I’ in solid, the Interface Segregation Principle, that we’ll look at in a future post.

We are done with the refactoring, at least as far as SRP is concerned. We can still take up other areas of improvement such as making the Order domain object cleaner by creating application services that will take care of the Checkout process. It may not be correct to put all these services in a single domain object, but it depends on the philosophy you follow in your domain design. That leads us to discussions on DDD (Domain Driven Design) which is not the scope of this post.

**Design patterns and practices in .NET: the Adapter Pattern**

April 25, 2013[6 Comments](http://dotnetcodr.com/2013/04/25/design-patterns-and-practices-in-net-the-adapter-pattern/#comments)

**Introduction**

The adapter pattern is definitely one of the most utilised designed patterns in software development. It can be used in the following situation:

Say you have a class – Class A – with a dependency on some interface type. On the other hand you have another class – Class B – that would be ideal to inject into Class A as the dependency, but Class B does not implement the necessary interface.

Another situation is when you want to factor out some tightly coupled dependency. A common scenario would be a direct usage of some .NET class, say HttpRuntime in a method that you want to test. You must have a valid HttpRuntime while running the test otherwise the test may fail and that is of course not acceptable. The solution is to let the method be dependent on some abstraction that is injected to it – the question is how to extract HttpRuntime and make it implement some interface instead? After all, it’s unlikely that you’ll have write access to the HttpRuntime class, right?

The solution is to write an adapter that sits between Class A and Class B that wraps Class B’s functionality. An example from the real world: if you travel from Britain to Sweden and try to plug in your PC in your hotel room you’ll fail as the electric sockets in Sweden are different from those in the UK. Class A is your PC’s power plug and Class B is the socket in the wall. The two objects clearly don’t fit but there’s a solution: you can insert a specially made adapter into the socket which makes the “conversion” between the plug and the socket enabling you to pair them up. In summary: the adapter pattern allows classes to work together that couldn’t otherwise due to incompatible interfaces. The adapter will take the form of an interface which has the additional benefit of extensibility: you can write many implementations of this adapter interface making sure your method is not tightly coupled to one single concrete implementation.

**Demo**

Open Visual Studio and create a new blank solution. Add a new class library called Domain and delete Class1.cs. Add a new class to this project called Customer. We’ll leave it void of any properties, that is not the point here. We want to expose the repository operations by an interface, so insert an interface called ICustomerRepository to the Domain layer:

|  |  |
| --- | --- |
| 1  2  3  4 | public interface ICustomerRepository  {  IList<Customer> GetCustomers();  } |

Add another class library called Repository and a class called CustomerRepository which implements ICustomerRepository:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | public class CustomerRepository : ICustomerRepository  {  public IList<Customer> GetCustomers()  {  //simulate database operation  return new List<Customer>();  }  } |

Add a new class library project called Service and a class called CustomerService to it:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | public class CustomerService  {  private readonly ICustomerRepository \_customerReposutory;  public CustomerService(ICustomerRepository customerRepository)  {  \_customerReposutory = customerRepository;  }  public IList<Customer> GetAllProducts()  {  IList<Customer> customers;  string storageKey = "GetAllProducts";  customers = (List<Customer>)HttpContext.Current.Cache.Get(storageKey);  if (customers == null)  {  customers = \_customerReposutory.GetCustomers();  HttpContext.Current.Cache.Insert(storageKey, customers);  }  return customers;  }  } |

This bit of code should be straightforward: we want to return a list of customers using the abstract dependency ICustomerRepository. We check if the list is available in the HttpContext cache. If that’s the case then convert the cached value and return the customers. Otherwise fetch the list from the repository and put the value to the cache.

So what’s wrong with the GetAllProducts method?

*Testability*

The method is difficult to test because of the dependency on the HttpContext class. If you want to get any reliable result from the test that tests the behaviour of this method you’ll need to somehow provide a valid HttpContext object. Otherwise if the test fails, then why did it fail? Was it a genuine failure, meaning that the customer list was not retrieved? Or was it because there was no HttpContext available? It’s the wrong approach making the test outcome dependent on such a volatile object.

*Flexibility*

With this implementation we’re stuck with the HttpContext as our caching solution. What if we want to change over to a different one, such as Memcached or System.Runtime? In that case we’d need to go in and manually replace the HttpContext solution to a new one. Even worse, let’s say all your service classes use HttpContext for caching and you want to make the transition to another caching solution for all of them. You probably see how tedious, time consuming and error-prone this could be.

On a different note: the method also violates the Single Responsibility Principle as it performs caching in its body. Strictly speaking it should not be doing this as it then introduces a hidden side effect. The solution to that problem is provided by the Decorator pattern, which we’ll look at in the next blog post.

**Solution**

It’s clear that we have to factor out the HttpContext.Current.Cache object and let the consumer of CustomerService inject it instead – a simple design principle known as Dependency Injection. As usual, the most optimal option is to write an abstraction that encapsulates the functions of a cache solution. Insert the following interface to the Service layer:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | public interface ICacheStorage  {  void Remove(string key);  void Store(string key, object data);  T Retrieve<T>(string key);  } |

I believe this is quite straightforward. Next we want to update CustomerService to depend on this interface:

|  |  |
| --- | --- |
| 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 | public class CustomerService  {  private readonly ICustomerRepository \_customerReposutory;  private readonly ICacheStorage \_cacheStorage;  public CustomerService(ICustomerRepository customerRepository, ICacheStorage cacheStorage)  {  \_customerReposutory = customerRepository;  \_cacheStorage = cacheStorage;  }  public IList<Customer> GetAllProducts()  {  IList<Customer> customers;  string storageKey = "GetAllProducts";  customers = \_cacheStorage.Retrieve<List<Customer>>(storageKey);  if (customers == null)  {  customers = \_customerReposutory.GetCustomers();  \_cacheStorage.Store(storageKey, customers);  }  return customers;  }  } |

We’ve got rid of the HttpContext object, so the next task is to inject it somehow using the ICacheStorage interface. This is the essence of the Adapter pattern: write an adapter class that will resolve the incompatibility of our home-made ICacheStorage interface and HttpContext.Current.Cache. The solution is really simple. Add a new class to the Service layer called HttpContextCacheStorage:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | public class HttpContextCacheStorage : ICacheStorage  {  public void Remove(string key)  {  HttpContext.Current.Cache.Remove(key);  }  public void Store(string key, object data)  {  HttpContext.Current.Cache.Insert(key, data);  }  public T Retrieve<T>(string key)  {  T itemsStored = (T)HttpContext.Current.Cache.Get(key);  if (itemsStored == null)  {  itemsStored = default(T);  }  return itemsStored;  }  } |

This is the adapter class that encapsulates the HttpContext caching object. You can now inject this concrete class into CustomerService. In the future if you want to make use of a different caching solution then all you need to do is to write another adapter for that: MemCached, Velocity, System.Runtime.Cache, you name it.

View the list of posts on Architecture and Patterns [here](http://dotnetcodr.com/architecture-and-patterns/).

<http://msdn.microsoft.com/en-us/magazine/dn683797.aspx>

<http://www.c-sharpcorner.com/UploadFile/akkiraju/s-o-l-i-d-design-principles-explained-part-1/>

<http://code.tutsplus.com/tutorials/solid-part-1-the-single-responsibility-principle--net-36074>

<http://thesoftwarearchitecture.wordpress.com/>

<http://www.codeguru.com/columns/experts/solid-principles-in-c-an-overview.htm>

<http://www.infoq.com/news/2013/08/solid-principles-revisited>