## Project Solid.SRP.Before

This project is a simple WPF application which uses MVVM pattern. On running this application, the user is presented with a canvas, or a dashboard, with three user controls: 1) a combo box control, containing a list of colors, 2) a list box control with can contain a more visual representation of the colors selected in combo box, and 3) an image control which changes its image based on selection of a color on the list box.

The three aforementioned user controls are implemented as different XAML “Views” contained in the “Views” folder. Each of these views, in its XAML code-behind, instantiates a corresponding “ViewModel” and assigns it to its public property referring to the same ViewModel class. The properties on each of the ViewModels are bound to the corresponding user elements in the XAML markup.

All “ViewModels” inherit from a Base ViewModel (BaseVM), which, in turn, implements INotifyPropertyChanged system interface. This ensures that all ViewModels can utilize the two-way binding infrastructure of XAML-based application.

ComboBoxVM, in its constructor, gets all color items (from Model part of the app), and assigns this collection to its AllColorItems property. When the user makes a selection in the ComboBox, an event is raised and captured; in the event handler for this event, we retrieve a reference to the ListBox control through the MainWindow object, and set the data context for the ListBox control to the list of colors after adding the current selected color of ComboBox.

ListBoxVM has similar logic to ComboBoxVM. The colors are added to ListBoxVM through selections in the ComboBox. However, when a color is selected in the ListView, an event is again raised and captured. This time we get a reference to the Image control, and refresh Image controls data context with the color item that was selected in the ListBox.

“Models” folder contains the ColorItem data class, and a Mock service for retrieving all color items. “Images” folder is self-explanatory.

The problem with this project is the tight coupling between the ViewModels and Views. We also have the atrocious problem of circular referencing between ViewModels and Views. Views depend on (or refer to) ViewModels, as can be seen in XAML code-behinds: View instantiates ViewModel. So far so good, but in the ViewModel code, ViewModel is obtaining a reference to, in a rather crude fashion, a View again (through the MainWindow object), thus resulting in circular references! The only way to retrieve a different ViewModel, in this example, is through the object tree of the Views.

This is a blatant violation of SRP!

## Project Solid.SRP.After

The “After” example is also a WPF application, with the same resultant functionality. This time we utilize “Mediator” pattern on top of MVVM pattern to achieve Single Responsibility. Please review Slide # 9 and 10 in the attached PowerPoint deck. Slide 9 and 10 are animated slides explain the mechanics of “Mediator” pattern. Slide 9 describes initial “Registration” process in Mediator pattern; Slide 10 describes “Notifications” in Mediator pattern.

Note: This project was inspired by a blogpost by Marlon from Thynk Software. See this link for more detail: <https://marlongrech.wordpress.com/2008/03/20/more-than-just-mvc-for-wpf>

By utilizing the Mediator pattern, on top of MVVM, we are able to abide by SRP.

## Project Solid.OCP

This project is a console application with two different namespaces: “Before” and “After.” Both namespaces have classes that accomplish a simple task of a) creating products, and b) shipping these products based on a selected “Shipping Method.” Shipping methods can be one of USPS, FedEx, DHL, UPS, etc.

“Before” namespace has classes that accomplish the aforementioned task, but they violate the “Open-Closed Principle” (OCP) in doing so. “After” namespace accomplishes the same task by abiding by the OCP.

**Before:**

1. ShippingMethod enum has four different values for Shipping Methods,
2. Product class defines a simple object with “Name” and “Price” properties, and a “Ship” method to ship the product based on total price charged to the customer,
3. ProductShipper class has methods:
   1. CalculateShippingCost, which calculates shipping cost based on ShippingMethod enum value,
   2. ShipProduct, which ships a product after calculating total cost by adding product price to shipping cost.
4. Driver class simply creates two Product objects and a ProductShipper object; it then uses the ProductShipper object to ship the two products.

**Problems with Before example:**

The “Before” namespace violates the “Open-Closed” principle. If we are required to add another Shipping Method in this example, we’ll have to:

1. Add a value to the ShippingMethod enum, and
2. Add a case statement to ProductShipper. CalculateShippingCost method to cater to the new Shipping Method.

These two classes (ShippingMethod and ProductShipper) are NOT closed for modifications, thus this example is a violation of OCP!

**After:**

1. Product class is the same as the one in “Before” example,
2. We got rid of the ShippingMethod enum,
3. The above enum was replaced by an interface, “IShippingStrategy,” and some concrete implementations of this interface,
4. ProductShipper class now takes an implementation of IShippingStrategy interface in its constructor, and assigns it to a private reference to the interface; it then uses the same reference to calculate shipping cost in ShipProduct method, and then ships the product with the sum of price and shipping cost.

**Abiding by the OCP:**

The “After” example abides by the OCP. The shipping cost calculations are implemented as a “Strategy” pattern. We got rid of the ShippingMethod enum, and the switch statement which calculated shipping cost based on an enum value. If we are required to add a shipping method, we would add a class that implements IShippingStrategy interface; none of the other classes need to be modified to accommodate the new shipping method. This example, therefore, is “Open” (implementation of IShippingStrategy interface) for extension, but “Closed” for modifications.

## Project Solid.LSP

This project is a console application as well with two different namespaces: “Before” and “After.” Both namespaces have classes that create a Square object and a Rectangle object, and then calculate areas for both of these objects.

“Before” namespace has classes that accomplish the aforementioned task, but they violate the “Liskov Substitution Principle” (LSP) in doing so. “After” namespace accomplishes the same task by abiding by the LSP.

**Before:**

1. Rectangle class is the parent class; it has virtual properties for Length and Width, and a virtual method to calculate area,
2. Square class inherits from Rectangle, overriding the virtual properties of Rectangle, assigning the same values in its setters to both Width and Length,
3. The Driver class instantiates Rectangle and Square object, and assigns them to their respective references; square object can be assigned to rectangle reference, as highlighted on line 56, thus achieving syntactic equivalence.

**Problems with Before example:**

Though the square object can be assigned to a rectangle reference, the invariants of rectangle do not hold after such an assignment. Invariants of rectangle entail modification of Length and Width properties independent of each other; this is not the case when square object gets assigned to rectangle reference, as setting the width or length property after such an assignment will overwrite both Length and Width to the same value (the overridden properties of Square assign the same values to both dimensions). Thus the Square object, in this example, is not substitutable for Rectangle, which is a violation of LSP.

**After:**

1. An abstract base class, Shape, is introduced that has an abstract method to calculate area,
2. Rectangle class inherits from Shape class now, adding its own two properties for Length and Width,
3. Square class, instead of inheriting from Rectangle, now inherits from Shape, adding its own single property, Dimension,
4. Driver program created the two objects, and calculates areas for these obejcts.

**Abiding by the LSP:**

The class inheritance structure is modified to break the relation between Rectangle and Square. A third abstract class, Shape, is introduced as the base class for both Rectangle and Square, thus allowing Rectangle and Square to assign/set their respective properties independent of each other. Now, the invariants of both Rectangle and Square hold true, thus abiding by the LSP.

## Project Solid.ISP

This project is also a console application with two different namespaces: “Before” and “After.” Both namespaces have classes that define relationship between Product and Order classes, and allows the driver program to ship an order, and print all items in an order.

“Before” namespace has classes that accomplish the aforementioned task, but they violate the “Interface Segregation Principle” (ISP) in doing so. “After” namespace accomplishes the same task by abiding by the ISP.

**Before:**

1. Product class defines a simple object with “Name” and “Price” properties, and a “Ship” method to ship the product based on total price charged to the customer,
2. IOrder is a bulky interface to be implemented on Order class, and defines methods utilized by Order class,
3. Order class implements IOrder interface, and contains (composite) a list of Product objects,
4. Driver class creates Product and Order, and uses methods on IOrder interface to Print and Ship order objects.

**Problems with Before example:**

The “Before” namespace violates ISP. IOrder is one bulky interface that has way too many responsibilities. The biggest problem with IOrder interface is that it contains Print method, which cannot be utilized by another class/entity like Customer, for instance. If we add a Customer class, we’ll have to define an ICustomer interface, and add another Print method to this new interface. This is clearly a violation of DRY principle (Don’t Repeat Yourself) as well as violation of ISP. In such a structure, we cannot achieve polymorphic printing behavior. For example, we cannot write something like the following.

List<IPrint> toBePrinted = new List<IPrint> { order, customer, vendor, stock };

foreach(var printable in toBePrinted)

{

printable.Print();

}

**After:**

1. PrintingStrategies.cs file contains implementation of “Strategy” pattern to print to console or a file,
2. ShippingStrategies.cs file also contains implementation of “Strategy” pattern, but to calculate the shipping cost,
3. Order.cs file contains two smaller interfaces: IOrder, which now has a single method, ShipAllItems, and IPrinter, with a single method of Print; the Order class now implements multiple, smaller interfaces,
4. OnlineOrder class is a specialization of Order class, which implements the interface method, Print, by adding one additional line to indicate that this is an OnlineOrder object,
5. Customer is an additional class added in “After” example, to highlight reusability (or DRY principle) of IPrinter.Print method,
6. Driver class creates different Order objects and prints them: placedOrder and customer objects are printed to the Console; newOrder object is printed to file, which can be found in “<ProjectDirectory>/bin/Debug/” folder.

**Abiding by the ISP:**

The “After” example indicates how, by segregating the bulkier IOrder interface into smaller interfaces, we can achieve reusability (printing the Customer entity by using the same IPrinter interface as Order class), thus abiding by DRY principle and ISP. Now we can achieve polymorphic printing behavior by writing the following code snippet. The only requirement for the following code to work is that order, customer, vendor, and stock objects all implement IPrint interface.

List<IPrint> toBePrinted = new List<IPrint> { order, customer, vendor, stock };

foreach(var printable in toBePrinted)

{

printable.Print();

}

## Project Solid.DIP

This project is also a console application with two different namespaces: “Before” and “After.” Both namespaces have classes that simulate information retrieval for certain public company stocks (Microsoft, Google, and Apple). The user inputs letters ‘M’, ‘G,’ or ‘A,’ and the program lets the user know whether the current stock price for the respective stock is higher than what was paid by the user. This comparison is done, in simulation, by comparing the stock price/info retrieved from a financial web service (e.g. Blomberg), and that retrieved from a local database.

“Before” namespace has classes that accomplish the aforementioned task, but they violate the “Dependency Inversion Principle” (DIP) in doing so. “After” namespace accomplishes the same task by abiding by the DIP.

**Before:**

1. StockWatcher class has methods for comparing stock prices, notifying the customer based on customer’s notification preference (Email or Text), and sending email and text messages to the customer,
2. Repositories.cs file has mock repositories that simulate retrieval of Stock object, based on stock symbol, from either an online service, or a local database,
3. Model.cs file contains data model classes such as Stock, Customer, CustomerNotificationPreference (enum). It also contains a static MockModel class which returns lists of stocks from database or service, current customer object, or RegisteredStockSymbols, all through static methods,
4. Driver program simulates, inside of a do-while loop, the retrieval and comparisons between online and database Stock objects.

**Problems with Before example:**

The “Before” namespace violates DIP. There are new() keywords in lower level modules, such as StockWatcher class. StockWatcher class instantiates new repository objects (on line 26 and 29); this is a violation of DIP because StockWatcher is a lower-level module as compared to the calling, Driver, class. A lower-level module should not create its own objects, but depend on abstractions which are assigned to by a higher-level module (Drive class in this example). The Driver class should create dependencies (objects) and inject those dependencies into the lower-level (StockWatcher) class when lower-level class is instantiated.

**After:**

1. StockMessageFactory.cs file contains an abstract class, MessageFactoryBase, and a concrete implementation of this abstraction, StockMessageFactory; StockMessageFactory now depends on abstractions: IStockRepository and IComparer. These abstract references are assigned to in the constructor of StockMessageFactory,
2. StockManager class also depends on abstractions, MessageFactoryBase and INotification; again, these abstract references are assigned to in the constructor of StockManager. NotifyCustomer method now uses these abstract references to simulate a) creating message for the customer, and b) notifying the customer based on his/her notification preference (Email or Text),
3. Repositories.cs file now has repositories that implement IStockRepository interface, thus providing a layer of abstraction,
4. Notifications.cs file utilizes “Strategy” pattern to provide, Email, Text or Postal notifications, thus introducing a layer of abstraction for any client of these notification classes,
5. Model.cs has the same static class, MockModel. We got rid of the CustomerNotificationPreference enum, and instead are using the notification strategies defined in Notifications.cs file. Customer entity now has only the name property. StockComparer class now implements IComparer<Stock> interface and provides comparison logic for Stock objects. Stock class is the same as in “Before” example.

**Abiding by the DIP:**

In the “After” example, there are no new() keywords in the lower-level modules! The only place, in the entire codebase for “After,” that we find new keywords is the Driver class. The Driver class, thus, serves as the “Composition Root” for the “After” application; driver class is responsible for creating all dependencies and then “injecting” these dependencies into the lower-level classes, at the time of creation of lower-level classes via constructors, thus abiding by the DIP. This can be referred to as the “poor man’s dependency inversion.” A more complex and fuller implementation of DIP, not implemented here, would entail offloading all object creation from the Driver class, and moving it to a “Dependency Factory,” or using a “Dependency Injection Container” (e.g. Managed Extensibility Framework, MEF).