# PTOM: The Dependency Inversion Principle

Posted by Jimmy Bogard on March 31, 2008

The Dependency Inversion Principle, or DIP, is often used interchangeably with Dependency Injection and Inversion of Control. However, following DIP does not mean we must automatically use a IoC container like Spring.NET, Windsor or StructureMap. IoC containers are tools to assist in applications adhering to DIP, but we can follow DIP without using IoC containers.

The Dependency Inversion Principle states:

- **High level modules should not depend upon low level modules. Both should depend upon abstractions.**
- **Abstractions should not depend upon details. Details should depend upon abstractions.**

The DIP can be a little vague, as it talks about "abstractions" but doesn't describe *what* is being abstracted. It speaks of "modules", which don't have much meaning in .NET unless you consider "modules" to be assemblies. If you're looking at Domain-Driven Design, modules mean something else entirely.

The Dependency Inversion Principle, along with the other SOLID principles, are meant to alleviate the problems of bad designs. The typical software I run into in existing projects has code organized into classes, but it still isn't easy to change. I usually see big balls of mud along with a crazy spider web of dependencies where you can't sneeze without breaking code on the other side of the planet.

## Spider webs and bad design

Bad designs and bad code is bad because it's hard to change. Bad designs are:

- Rigid (change affects too many parts of the system)
- Fragile (every change breaks something unexpected)
- Immobile (impossible to reuse)

Some people's ideas of "bad design" would be something like seeing string concatenation instead of a StringBuilder. While this may not be the best performing choice, string concatenation isn't necessarily a bad design.

It's pretty easy to spot bad designs. These are sections of code or entire applications that you dread touching. A typical example of rigid, fragile and immobile (bad) code would be:

```
public class OrderProcessor
{
    public decimal CalculateTotal(Order order)
    {
        decimal itemTotal = order.GetItemTotal();
        decimal discountAmount = DiscountCalculator.CalculateDiscount(order);

        decimal taxAmount = 0.0M;

        if (order.Country == "US")
            taxAmount = FindTaxAmount(order);
        else if (order.Country == "UK")
            taxAmount = FindVatAmount(order);

        decimal total = itemTotal - discountAmount + taxAmount;

        return total;
    }

    private decimal FindVatAmount(Order order)
    {
        // find the UK value added tax somehow
        return 10.0M;
    }

    private decimal FindTaxAmount(Order order)
    {
        // find the US tax somehow
        return 10.0M;
    }
}
```

The OrderProcessor sets out to do something very simple: calculate the total of an Order.  To do so, it needs to know the item total of the order, any discounts applied, as well as the tax amount (which depends on the Order's Country).

**Too many responsibilities**

To see why the DIP goes hand-in-hand with the Single Responsibility Principle, let's list out the responsibilities of the OrderProcessor:

- Knowing how to calculate the item total
- Finding the discount calculator and finding the discount
- Knowing what country codes mean
- Finding the correct taxing method for each country code
- Knowing how to calculate tax for each country (commented out for brevity's sake)
- Knowing how to combine all of the results into the correct final total

If a single class (or a single method in this case) answers too many questions (how, where, what, why etc.), it's a good indication that this class has too many responsibilities.

To move towards a good design, we need to remove the external dependencies of this class to pare it down to its core responsibility: finding the order total.  Offhand, the dependencies I see are:

- DiscountCalculator
- Tax decisions

In the future, we might need to support more countries, which means more tax services, and more responsibilities.  To reduce the rigidity, fragility and immobility of this design, we need to move these dependencies outside of this class.

**Towards a better design**

When following the DIP, you notice that the Strategy pattern begins to show up in a lot of your designs.  Strategy tends to solve the "details should depend on abstractions" part of the DIP.  Factoring out the DiscountCalculator and the tax decisions, we wind up with two new interfaces:

- IDiscountCalculator
- ITaxStrategy

I'm not a huge fan of the "ITaxStrategy" name, but it will suffice until we find a better name from our model.

**Factoring out the dependencies**

To factor out the dependencies, first I'll create a couple of interfaces that match the existing method signatures:

```
public interface IDiscountCalculator
```

```
{
    decimal CalculateDiscount(Order order);
}

public interface ITaxStrategy
{
    decimal FindTaxAmount(Order order);
}
```

Now that I have a couple of interfaces defined, I can modify the OrderProcessor to use these interfaces instead:

```
public class OrderProcessor
{
    private readonly IDiscountCalculator _discountCalculator;
    private readonly ITaxStrategy _taxStrategy;

    public OrderProcessor(IDiscountCalculator discountCalculator,
                          ITaxStrategy taxStrategy)
    {
        _taxStrategy = taxStrategy;
        _discountCalculator = discountCalculator;
    }

    public decimal CalculateTotal(Order order)
    {
        decimal itemTotal = order.GetItemTotal();
        decimal discountAmount = _discountCalculator.CalculateDiscount(order);

        decimal taxAmount = _taxStrategy.FindTaxAmount(order);

        decimal total = itemTotal - discountAmount + taxAmount;

        return total;
    }
}
```

The CalculateTotal method looks much cleaner now, delegating the details of discounts and tax to the appropriate abstractions. Instead of the OrderProcessor depending directly on details, it depends solely on the abstracted interfaces we created earlier. The specifics of how to find the correct tax method is now encapsulated from the OrderProcessor, as is the hard dependency on a static method in the DiscountCalculator.

**Filling out the implementations**

Now that we have the interfaces defined, we need actual implementations for these dependencies. Looking at the DiscountCalculator, which is a static class, I find that I can't immediately change it to a non-static class. There are many other places with references to this DiscountCalculator, and since it's the real world, none of these other places have tests.

Instead, I can just use the Adapter pattern to adapt the interface I need for an IDiscountCalculator:

```
public class DiscountCalculatorAdapter : IDiscountCalculator
{
    public decimal CalculateDiscount(Order order)
    {
        return DiscountCalculator.CalculateDiscount(order);
    }
}
```

In applying the Adapter pattern, I just wrap the real DiscountCalculator in a different class.  The advantage of the Adapter pattern in this case is that the existing DiscountCalculator can continue to exist, but ~~if~~ when the mechanism for calculating discounts changes, my OrderProcessor does not need to change.

For the tax strategies, I can create two implementations for each kind of tax calculation being used today:

```
public class USTaxStrategy : ITaxStrategy
{
    public decimal FindTaxAmount(Order order)
    {
    }
}

public class UKTaxStrategy : ITaxStrategy
{
    public decimal FindTaxAmount(Order order)
    {
    }
}
```

I left the implementations out, but I basically moved the methods from the OrderProcessor into these new classes.  Neither of the original methods used any instance fields, so I could copy them straight over.

My OrderProcessor now has dependencies factored out, so its single responsibility is easily discerned from looking at the code.  Additionally, the implementations of IDiscountCalculator and ITaxStrategy can change without affecting the OrderProcessor.

**Isolating the ugly stuff**

For me, the DIP is all about isolating the ugly stuff.  For calculating order totals, I shouldn't be concerned about where the discounts are or how to decide what tax strategy should be used.  We did increase the number of classes significantly, but this is what happens when we move away from a procedural mindset to a true object-oriented design.

I still have the complexity to solve of pushing the dependencies into the OrderProcessor.  Clients of the OrderProcessor now have the burden of creating the correct dependency and giving them to the OrderProcessor.  But that problem is already solved with Inversion of Control (IoC) containers like Spring.NET, Windsor, StructureMap, Unity and others.

These IoC containers let me configure the "what" when injecting dependencies, so even that decision is removed from the client.  If I didn't want to go with an IoC container, even a simple creation method or factory class could abstract the construction of the OrderProcessor with the correct dependencies.

By adhering to the Dependency Inversion Principle, I can create designs that are clean, with clearly defined responsibilities.  With the dependencies extracted out, the implementation details of each dependency can change without affecting the original class.

And that's my ultimate goal: code that is easy to change.  Easier to change means a lower total cost of ownership and higher maintainability.  Since we know that requirements will eventually change, it's in our best interest to promote a design that facilitates change through the Dependency Inversion Principle.

← Stop creating custom delegate types Reacting to change →