Inversion of Control

SOLID | DIP | IoC | DI | IoC Containers | Made Easy

23 March 2015

Shuhab-u-Tariq
www.shuhab.com

Table of Contents

Chapter 1: SOLID Principles of Object Oriented Design

The SOLID principles are fundamental to designing effective, maintainable, object-oriented systems. When used appropriately, these principles can improve the coupling of your application, making it more robust, scalable, extensible, non-fragile and testable in the face of changing requirements. SOLID is an acronym of the following:

- Single Responsibility Principle
- Open / Closed Principle
- Liskov Substitution Principle
- Interface Segregation Principle
- Dependency Inversion Principle
- Don’t Repeat Yourself Principle

The Single Responsibility Principle (SRP)

The Single Responsibility Principle states that every object should have a single responsibility, and that responsibility should be entirely encapsulated by the class. Robert C. Martin, sums this up by saying, “there should never be more than one reason for a class to change.”

Single Responsibility Principle
Just because you can, doesnn't mean you should

Cohesion means how strongly-related and focused the various responsibilities of a module are. Coupling is the degree to which each program module relies on each one of the other modules. SRP strives for low coupling and high cohesion.

Responsibilities are defined as axes of change. Requirement changes typically map to responsibilities. Responsibilities are directly proportional to the likelihood of change. Having multiple responsibilities within a class couples together these responsibilities. The more classes a change affects, the more likely the change will introduce errors. Thus it is important to try and craft classes in such a way that the areas that are most likely to change are encapsulated into separate classes with single responsibilities.

The Open / Closed Principle (OCP)

The Open/Closed Principle states that software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification.

Here "Open for extension" means, we need to design our module/class in such a way that the new functionality can be added only when new requirements are generated. "Closed for modification" means change to source or binary code is not required. There shouldn't be need to recompile the existing pieces of the application. It means that once we have developed a class that has gone through unit-testing, we should then not alter it until we find bugs. For a class to obey OCP (change behaviour without changing source code), it has to rely on abstractions. In .NET, abstractions include “Interfaces” & “Abstract Base Classes”. Some of the approaches that can be used to achieve OCP are:

Parameters (Procedural Programming)

- Allow client to control behaviour specifics via a parameter
- Combined with delegates/lambda can be very powerful approach

Inheritance / Template Method Pattern

- Child types override behaviour of a base class (or interface)

Composition / Strategy Pattern

- Client code depends on abstraction
- Provides a “plug-in” model
- Implementations utilize Inheritance; Client utilizes Composition

Note: If you know from your own experience in the problem domain that a particular class of change is likely to recur, you can apply OCP up front in your design. Practically, in real time scenarios, no design can be closed against all changes. So don’t apply OCP at first, as it adds complexity to design. If the module changes once, accept it. However once it changes a second time, now you're at the point where you know this is something that's volatile. It has shown that it has a propensity for frequent change. It's time to refactor it to achieve the Open/Closed Principle. And the way you do that is by finding an abstraction, creating an interface, and then extracting out the “if else” logic, or switch statement logic into separate classes where each one represents a particular node in that decision tree. Conformance to OCP yields flexibility, reusability and maintainability.

The Liskov Substitution Principle (LSP)

The Liskov Substitution Principle states that “subtypes must be substitutable for their base types.” In order for substitutability to work, child classes must not remove base class behaviour, nor should they violate base class invariants. In general, the calling code should not know that there is any difference at all between a derived type and its base type. In order to follow LSP, derived classes must not violate any constraints defined (or assumed by clients) on the base classes.

Note: Invariants consist of reasonable assumptions of behaviour by clients. Invariants can be expressed as pre-conditions and post-conditions for methods. There is a technique called “Design by Contract” that makes defining these pre and post conditions explicit within the code itself.

Conformance to LSP allows for proper use of polymorphism and produces more maintainable code.

The Interface Segregation Principle (ISP)

The Interface Segregation Principle (ISP) states that “clients should not be forced to depend on methods they do not use”. A corollary to this would be to prefer small, cohesive interfaces to ‘fat’ interfaces. ISP violations result in classes that depend on things they don’t need, thereby increasing coupling, and reducing flexibility & maintainability. So if a class depends on a ‘fat’ interface:

- Create a smaller interface with only what is required
- Have the fat interface implement the new interface
- Reference this new interface within the code

The client code shouldn’t depend on things it doesn’t need. So keep interfaces lean and focused. Large interfaces should be refactored so they inherit smaller interfaces.

The Dependency Inversion Principle (DIP)

High-level modules should not depend on low level modules through direct instantiations or static method calls. Both should depend on abstractions. Class dependencies should be declared explicitly in their constructors. Inject dependencies via constructor, property, or parameter injection. The principle states that:

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

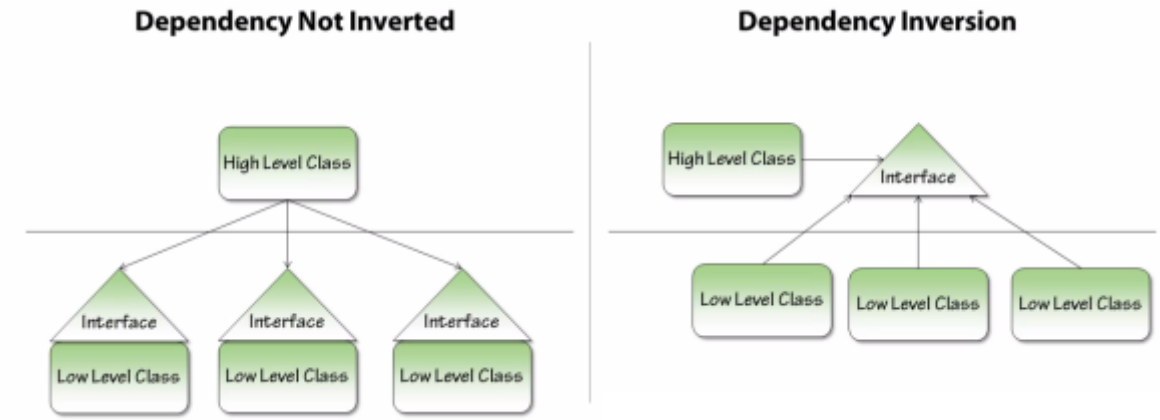
(Discussed in detail in Chapter 2)

The Don't Repeat Yourself Principle (DRY)

The Don't Repeat Yourself Principle states that every piece of knowledge must have a single, unambiguous representation in the system. Repetition in logic calls for abstraction. Repetition in process calls for automation. Code must be refactored to remove repetition that breeds errors and waste.

Chapter 2: Dependency Inversion

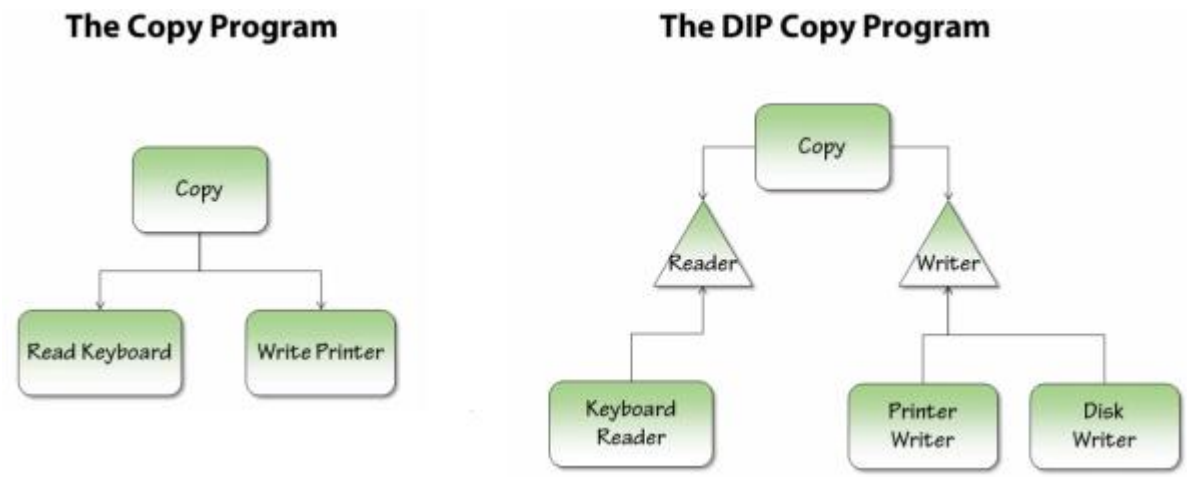
Instead of lower level modules defining an interface that higher level modules depend on, higher level modules define an interface that lower level modules implement.



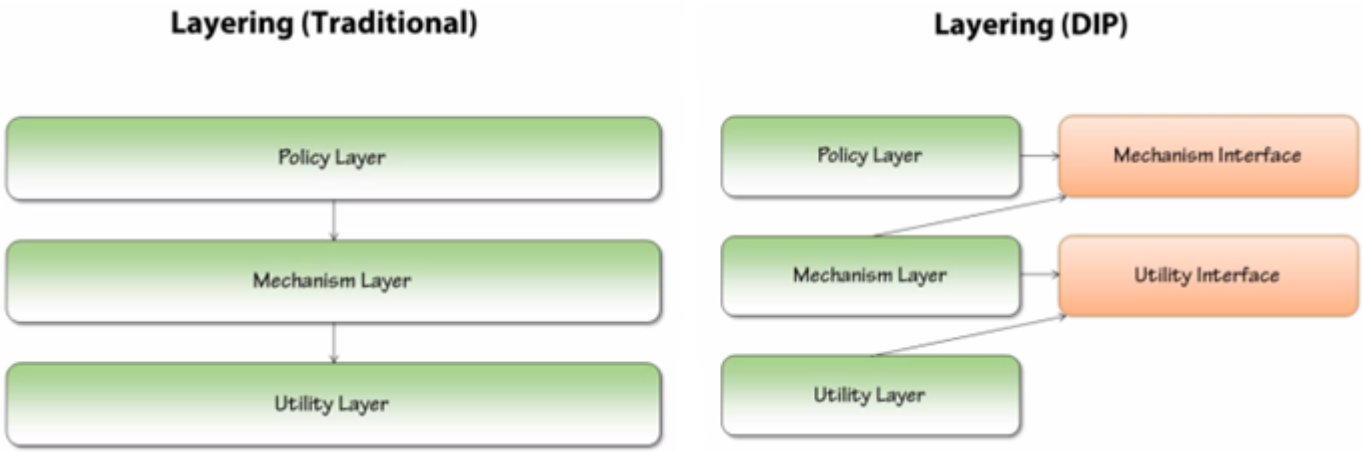
Dependency Inversion Principle *(From Bob Martin's paper on DIP)*

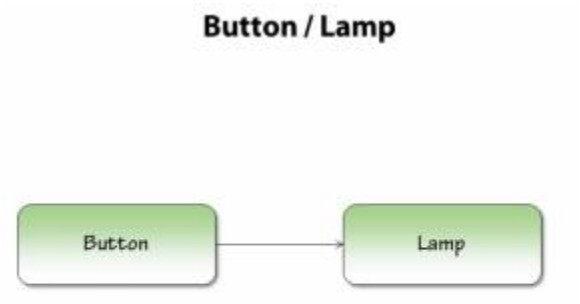
Principle used in architecting software. It states:

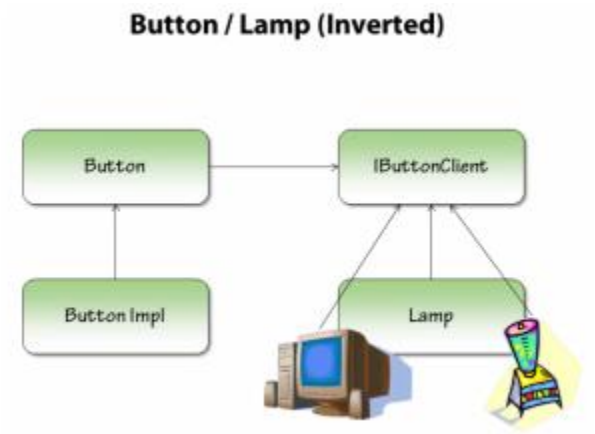
- High-level modules should not depend on low level modules. Both should depend on abstractions.
- Abstractions should not depend upon details. Details should depend upon abstractions.



Uncle Bob's Layering Example (From Bob Martin's paper on DIP)







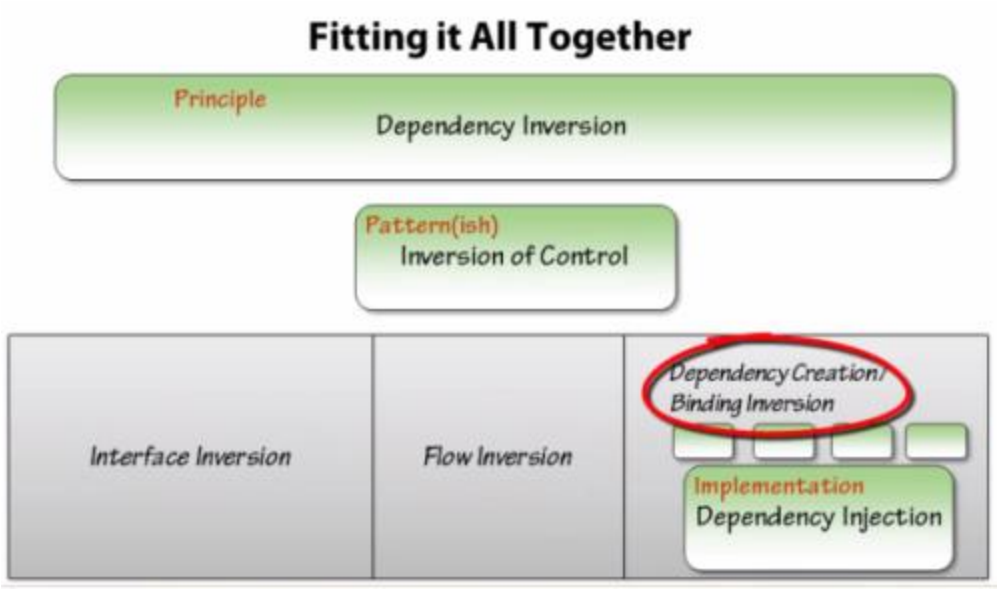
When Dependency Inversion Principle is applied, it means the high level classes are not working directly with low level classes. They are using interfaces as an abstract layer. In this case instantiation of new low level objects inside the high level classes (if necessary) cannot be done using the operator new. Instead, some of the Creational design patterns can be used, such as Factory Method, Abstract Factory, or Prototype. The Template Design Pattern is an example where the DIP principle is applied. Using this principle implies an increased effort, and will result in more classes and interfaces to maintain, in a few words in more complex code, but more flexible. This principle should not be applied blindly for every class or every module. If there is a class functionality that is more likely to remain unchanged in the future, there is not a need to apply this principle.

Chapter 3: Inversion of Control

It is a specific pattern used to invert interfaces, flow & dependencies. IoC gives ways to implement DIP (dependency inversion principle). It is sort of a high-level pattern that can be applied in different ways to invert different kinds of controls. It gives us:

- Control over the interface between two systems or components.
- Control over the flow of an application.
- Control over the dependency creation and binding.

In DIP, we subscribed to the idea that high level modules should not depend on low level modules. DIP says that changes in low level modules should not affect high level modules. But it doesn't say how to do it. Inversion of Control (IoC) is a way of implementing Dependency Inversion, but it doesn't provide a specific implementation.

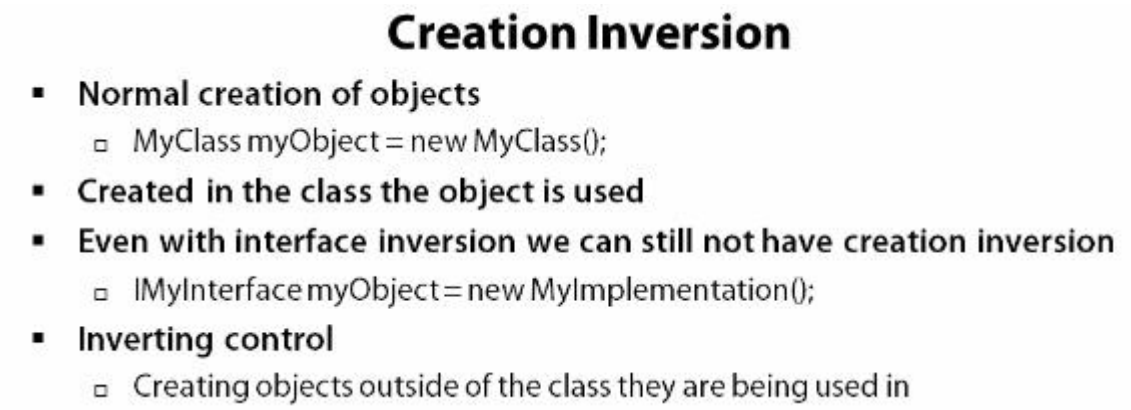


There are different ways to implement IoC (viz. Interface Inversion, Flow Inversion, Creation Inversion), most popular being the Creation Inversion.

Note: Interface is an externally exposed way to interact with something. An interface should have more than one implementation; otherwise it doesn't really serve much of a purpose.

Creation Inversion

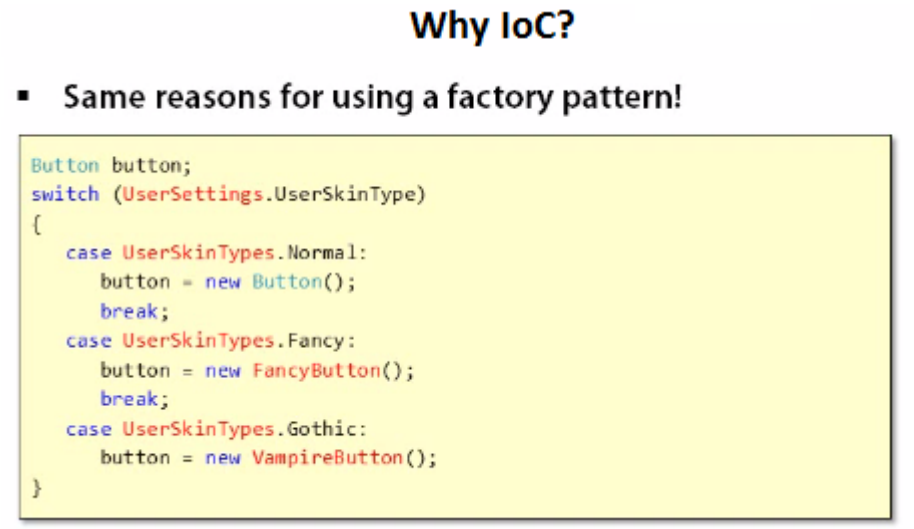
So the normal way that you would create objects if you're just programming, creating your class, is that you will just new up that object inside of your class. So typically you would create an object inside of the class that's going to use that object. That's normal creation; the control is in the class using the dependency. Even if you use an interface, even if you inverted the interface, etc.



So even if you've inverted the interface, even if you've inverted the dependency, if you're still creating the object inside of the class that's using it, you still have a dependency. Because that, if you imagine that you're class is a high level module or high level class, and it still has to new up the lower level class, well changes to that lower level class are still going to impact that higher level class. That higher level class still depends on the lower level class. This is where "Creation Inversion" finds its place.

So, to invert this control of creation and break the binding dependency, we have to create the objects that are used by the class outside of itself. Somehow we have to make it so that the creation is not happening in the high level module or the high level class that's depending on the lower level module. We need something or some other way to bind these two together. So we need to invert that control of creation, take away the reins from our class and give it to someone else.

So why would you do IoC? As discussed earlier, the inversion of control is done to correctly apply the principle of dependency inversion. In order to achieve IoC, there should be some way of totally breaking the dependencies. Some of the reasons for doing this are a lot like the reasons of implementing the factory pattern. It's the same kind of idea here where the creation of some set of objects resides in one central place. We try to move the dependency out by changing where the creation of object happens.



In the above example, there is some kind of UI, trying to create a button. User could skin a button with different kinds of button styles. For a normal skin-type case, it creates a new button. In case of a fancy skin-type, it creates a new fancy button. And for a gothic skin-type, it creates a vampire button. And it could go and on, but if you think about it, everywhere that you create a button in your code, if you have the control not inverted, you're going to have the same kind of switch statement. After you have created 10 screens of your user interface all using maybe 20 buttons, you're going to have this code copy and pasted in a lot of places. So this is a good reason to use an inversion of control, a creational inversion of control. We really want to move the creation of this button to somewhere central and common so that when the types of buttons change in the system, we don't have to change 50 places in the code.

So let's look at what would happen if we inverted this creation. So we're going to move this button creation from our class into a factory, a button factory. The code has now simplified down to one line where we're just declaring a button and then we have a button factory on which we are dependent for creation of the button. On this button factory, we just call create button and we assume here that this button factory has access to the user setting to know user's selection of the skin-type which helps us o decide what kind of button to return. So we don't need to know anymore what type of button. We just care that it implements this button interface or inherits from this abstract button. And then we can use it just like a button and we don't actually need to know what type of button it is. Instead, to this button factory, we just say, give us a button and it says okay here's a button. We don't need to know what type it is. So this is how we've inverted this creation dependency and then we've gained a huge value out of it here.

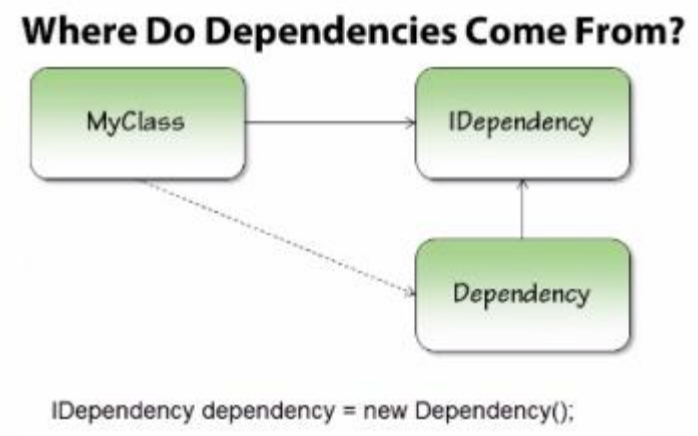
Types of Creation Inversion

Basically anytime that you are taking the creation of a dependency or the binding of that dependency and you're moving it outside of the class, you are inverting the control of the creation of that dependency. And so anyway that you can think to do that, that makes sense, is going to qualify it as creation inversion.

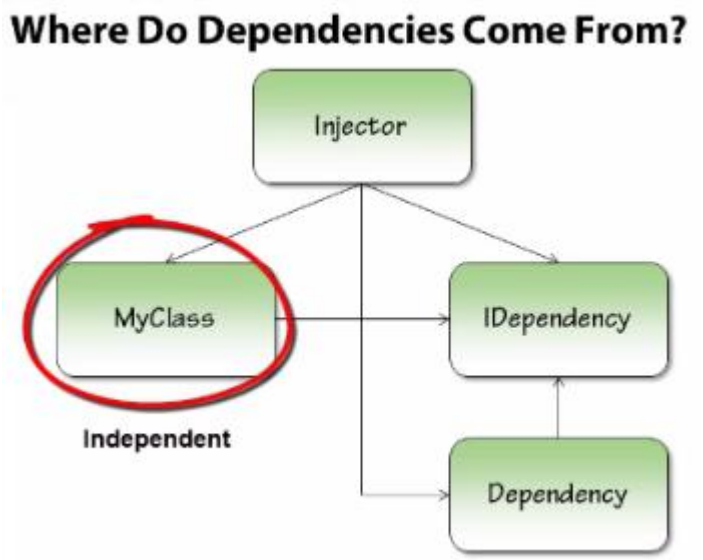
Chapter 4: Dependency Injection (DI)

DI is the implementation of IoC to invert dependencies. It is a type of IoC where we move the creation and binding of a dependency outside of the class that depends on it. [Another IoC type is the Service Locator Pattern]

In the diagram here, we've done the basic inversion of control or dependency inversion principle. MyClass is depending on IDependency, and we have a dependency that implements that IDependency. So what we've done here is we've basically decoupled our class directly from the dependency. It is no longer depending directly on that dependency.



So, in this case here (without dependency injection), even though we've decoupled my class from the concrete dependency, it still is going to create that dependency. Even though my class references only IDependency, but how does that dependency get instantiated. Well if that dependency isn't passed in some way or provided to the class, then it stands to reason that my class must create that dependency. So somewhere in my class, by default, it's going to have a new dependency. It may have a variable of type IDependency, but it is going to have some kind of statement like IDependency dependency = new dependency. And so to answer the question, in this case, dependencies come from our class, our dependent class.



We can change this behaviour and this is where dependency injection comes into play. In DI, we have another class that's acting as an injector. This injector can inject the dependency in a variety of ways. As we can see in the diagram, this injector knows about my class, it knows about the interface of IDependency, and it knows about the dependency. And what this injector is going to do is that it binds these things together. So injector's responsibility is to instantiate my class and then it will instantiate the dependency and it will pass the dependency into my class. Regardless of how it does it, the key concept here is that when we're using dependency injection, there has to be something that's doing the injection. And that injector is going to depend on all of the components and make it so that your class can truly be independent of that dependency. So, whereas in the former example, dependencies came from MyClass and MyClass had to do new dependencies; now after using dependency injection, dependencies come from the injector.

Constructor Injection *(most popular form of DI)*

- Pass dependency into dependent class via constructor.

```
ICreditCard creditCard = new MasterCard();
Shopper shopper = new Shopper(creditCard);

public class Shopper
{
   private readonly ICreditCard creditCard;

   public Shopper(ICreditCard creditCard)
   {
      this.creditCard = creditCard;
   }
}
```

Property (or Setter) Injection

- Create a setter on the dependent class. Then use the setter to set the dependency.

```
ICreditCard creditCard = new MasterCard();
Shopper shopper = new Shopper();
shopper.CreditCard = creditCard;

public class Shopper
{
   public ICreditCard CreditCard { get; set; }
}
```

Interface Injection

- Dependent class implements an interface. Injector uses the interface to set the dependency.

```
ICreditCard creditCard = new MasterCard();
Shopper shopper = new Shopper();
((IDependOnCreditCard)shopper).Inject(creditCard);

public class Shopper : IDependOnCreditCard
{
   private ICreditCard creditCard;
   public void Inject(ICreditCard creditCard)
   {
      this.creditCard = creditCard;
   }
}

public interface IDependOnCreditCard
{
   void Inject(ICreditCard creditCard);
}
```

DI Caution!

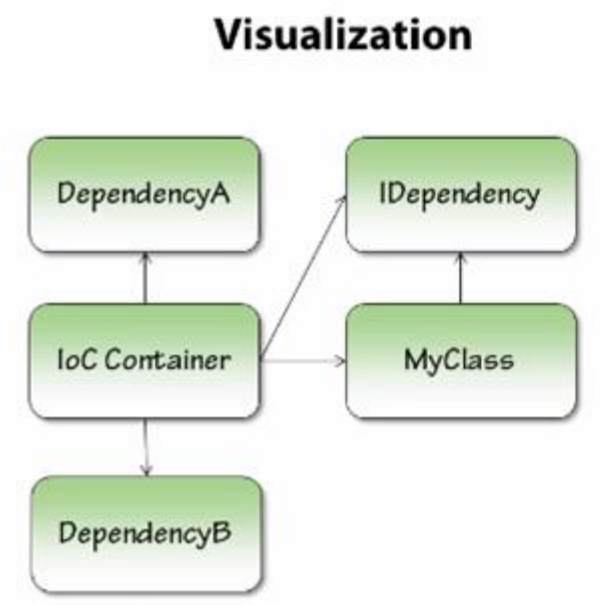
- **Leaks the internal implementation details of a class**
  - Violates encapsulation
  - Injecting "guts" into class
- **Prevents deferred creation**
  - Dependencies created before needed
  - Watch out for large object graphs
- **Numbs you from the pain**
  - Easier to unit tests classes that should be broken up
  - Watch out for too many dependencies

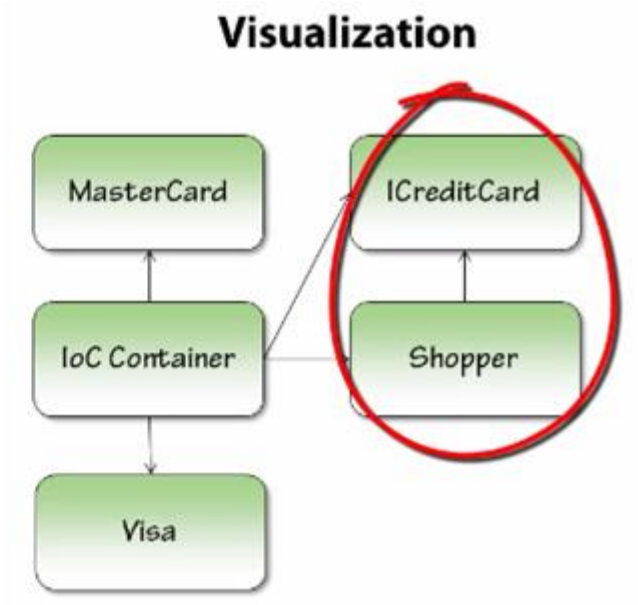
Chapter 5: Building an IoC Container

- What is an IoC Container?
- Manual Constructor Injection
- Resolving Dependencies
- Creating the Container

What is an IoC Container?

- A framework for doing dependency injection.
- Configure dependencies
- Automatically resolves configured dependencies





MasterCard & VisaCard (concrete classes) implement ICreditCard (interface). In Shopper class contructor, we manually inject credit card.

```
public interface ICreditCard
{
    string Charge();
}
```

```
public class VisaCard : ICreditCard
{
    public string Charge()
    {
        return "Swiping the VisaCard";
    }
}

public class MasterCard : ICreditCard
{
    public string Charge()
    {
        return "Swiping the MasterCard";
    }
}
```

```
public class Shopper
{
    private readonly ICreditCard creditCard;

    public Shopper(ICreditCard creditCard)
    {
        this.creditCard = creditCard;
    }

    public void Charge()
    {
        var chargeMessage = creditCard.Charge();
        Console.WriteLine(chargeMessage);
    }
}
```

Manual constructor dependency injection *(Poor man's DI)*

```
class Program
{
    static void Main(string[] args)
    {
        ICreditCard creditCard = new MasterCard();
        //ICreditCard otherCreditCard = new VisaCard();

        var shopper = new Shopper(creditCard);
        //var shopper = new Shopper(otherCreditCard);

        shopper.Charge();

        Console.Read();
    }
}
```

Resolving Dependencies

Now let's try and take this one step further, closer to a container. One problem with the above code is that we have to decide before we create our class what the concrete type is going to be. So this isn't very configurable and it's not very smart. Even though we are manually injecting the dependencies, it's not resolving it for us.

Let's rewrite this. Instead of passing in concrete dependency to Shopper [var shopper = new Shopper(creditCard)], we 'll use a resolver, and we will call resolve credit card. The method will now resolve our credit card and we don't have to bind it. There is a specific responsibility inside the resolver to resolve credit card. This makes it a little bit configurable because we could implement this in a variety of different ways. We could read from the config file that says, the current credit card is this, etc. We could look at some kind of user setting to see or what the credit card should be. We could use some business rules or some logic here to decide. So let's comment out our old code and let's get this working.

```
class Program
{
    static void Main(string[] args)
    {
        //ICreditCard creditCard = new MasterCard();
        //ICreditCard otherCreditCard = new VisaCard();

        //var shopper = new Shopper(creditCard);
        //var shopper = new Shopper(otherCreditCard);

        Resolver resolver = new Resolver();
        var shopper = new Shopper(resolver.ResolveCreditCrad());

        shopper.Charge();

        Console.Read();
    }
}
```

```
public class Resolver
{
    public ICreditCard ResolveCreditCrad()
    {
        if (new Random().Next(2) == 1)
        {
            return new VisaCard();
        }

        return new MasterCard();
    }
}
```

Inside Resolver class, there is a method to ResolveCreditCard. For demonstration purpose, it's using a Random number generator (less than 2). So it is giving either zero or one. If it is one, then return a new VisaCard, otherwise return a new MasterCard. So it is kind of giving a 50/50 chance, randomizing what we return here. In real time scenario, this will get replaced by some sort of business logic (e.g. determining what to return based on reading from config file, etc). When this code is executed, it will randomly pick between master card & visa card. But we're not worrying about the resolution here in our code. We can just use this resolver and we can kind of throw this resolver wherever we want and whenever we have dependencies to resolve. It's like a factory. As you can see, here it behaves more like a service locator pattern.

Creating the IoC Container

Let's now take this a step further and let's see how we would actually turn this into more of a real container. Ideally, the main objective is to resolve a whole dependency tree. We don't want to have everywhere in our code where we need a new type that we have to call resolver and then have a method to resolve that type. So we really want to do a different syntax and let's look at how we would do that.

```
public class Resolver
{
    private Dictionary<Type, Type> dependencyMap =
        new Dictionary<Type, Type>();

    public T Resolve<T>()
    {
        return (T)Resolve(typeof(T));
    }

    private object Resolve(Type typeToResolve)
    {
        Type resolvedType = null;

        try
        {
            resolvedType = this.dependencyMap[typeToResolve];
        }
        catch
        {
            throw new Exception(string.Format("Could not resolve type {0}",
                typeToResolve.FullName));
        }

        var firstConstructor = resolvedType.GetConstructors().First();
        var constructorParameters = firstConstructor.GetParameters();

        if (constructorParameters.Count() == 0)
        {
            return Activator.CreateInstance(resolvedType);
        }

        IList<object> parameters = new List<object>();

        foreach (var parameterToResolve in constructorParameters)
        {
            parameters.Add(this.Resolve(parameterToResolve.ParameterType));
        }

        return firstConstructor.Invoke(parameters.ToArray());
    }

    public void Register<TFrom, TTo>()
    {
        this.dependencyMap.Add(typeof(TFrom), typeof(TTo));
    }
}
```

The resolve method checks whether the type requested is present in the dictionary or not. In our case, this could be ICreditCard that we pass in and then we have mapped this to VisaCard. Then we check for constructors. In this case, the parameters would be zero and so, we'll just create a VisaCard. In the case that we do have parameters, we're going to have to handle this a bit differently.

The code makes a list of all parameters as it creates them and then constructs the object with the parameter list. So, for each parameter to resolve in constructor parameters, it loops through each constructor parameter, as it needs to resolve each one, and add them to the list.

We're going to do parameters.Add and then we're going to recursively call our Resolve method to resolve the types. So we need to grab our parameter to resolve and then figure out, okay what is the actual parameter type? And there's a method on here that tells us what that type is. So for each parameter of that constructor, figure out what the type is, and then resolve the type. This will kind of chain out like a tree until we get to these root nodes where they don't have any parameters in the constructor and then you'll just hit this Activate code and activate them. Once we have the whole list, we just need to create our object using that list. To return a new type that we're creating, we're not going to use activator in this case, because we have some parameters here.

We're going to use this other method on the type called “Invoke”. So the way we'll do this is we'll say firstConstructor.Invoke and this has overload that just takes the parameters. And this should do it. We have created a dependency injection; an injector or another way of saying that is an IoC container.

```
class Program
{
    static void Main(string[] args)
    {
        Resolver resolver = new Resolver();

        resolver.Register<Shopper, Shopper>();
        //resolver.Register<ICreditCard, MasterCard>();
        resolver.Register<ICreditCard, VisaCard>();

        var shopper = resolver.Resolve<Shopper>();

        shopper.Charge();

        Console.Read();
    }
}
```

So the last thing that we need to do is to set up these dependencies. So let's go ahead and register our types. So here what we'll do is before we use shopper, we're going to go to our resolver and we're going to register and then we're just going to pass the type that we want to register (e.g. TFrom: shopper to TTo: shopper). So when we ask for a shopper, give us a shopper. We could have created an IShopper interface, but that'd be a waste because we don't have any other implementations of it. So in that case I'm just mapping shopper to itself, but this is important because this resolver, this IoC container that we've created, needs to know how to do that. So let's create this register method. And in this register method, I'm going to call TFrom and TTo. All it's going to do is just add that to our dictionary so we have a dependency map. So, in our case, TFrom will be ICreditCard, and TTo will be either a MasterCard or Visa card. And it's going to put it in that dictionary so we can look it up. Now you can see the power of what we're doing here. Once we've set up these rules of how to resolve things, this resolver (IoC container), that we've created, is going to do it for us. So we're just calling resolver.Resolve on shopper and while it looks at the shopper, it will need a credit card. It will then determine that the type registered for credit card is a VisaCard. And it's going to resolve that.

To summarize what happens here, we registered our types, so they go into the dictionary that says when you ask me for this type (TFrom), I will give you this type (TTo). And then we ask for a shopper, so we call resolver.Resolve<Shopper>(). Then this resolve method checks if that type is registered in our map and whether it knows to return a card of type (say Visa) when someone asks for ICreditCard. If it's not registered, it throws an exception. If it is, then it grabs the constructor. It checks the parameters, and if there are no parameters, then it just simply activates it, creates a new instance and sends it back. If there are

parameters, then it needs to resolve each one of those first. So, it loops through each parameter and calls the resolve method and checks if it can construct each one of those. And so it gets the list of parameters, and calls the constructor passing in that list of parameters.

There are various professional, commercial, and open-source frameworks out there for doing dependency injection (IoC containers). And they do a lot more than our example. For instance, they manage lifetime of the objects and things like that, etc. These frameworks use the technique of dependency injection to break applications into loosely-coupled, highly cohesive components, and then glue them back together in a flexible manner. Some of the frameworks used for dependency injection are:

- Microsoft Unity (part of Enterprise Library. Licenced as MS-PL)
- Castle Windsor (based on the Castle MicroKernel. Licenced under Apache2)
- Ninject (formerly Titan. Licenced under Apache2)
- StructureMap (Licenced under Apache2)
- Spring.NET (Licenced under Apache2)
- Autofac (Licenced under MIT)
- LinFu (Licenced under Lesser GPL)

Recommended Reading

- Robert C. Martin, Micah Martin (2006) Agile Principles, Patterns, and Practices in C#
- Erich Gamma, John Vlissides, Ralph Johnson, Richard Helm (1994) Design Patterns: Elements of Reusable Object-Oriented Software
- Martin Fowler (2004) Inversion of Control Containers and the Dependency Injection pattern [online]. Available at: http://www.martinfowler.com/articles/injection.html