AUTODESK **UNIVERSITY** 2014

Sharpen Your C# Code for Autodesk® AutoCAD®

Scott McFarlane – Woolpert, Inc.

SD5215 This class is for the Autodesk AutoCAD C# developer who wants to improve code quality and maintainability. Learn how to apply good software design patterns to your AutoCAD code, and take full advantage of the advanced features of the C# language. We cover topics such as the single responsibility principle, reducing duplicate code, abstraction, and dependency injection. We touch on WPF and the Model-View-ViewModel pattern. We also explore C# features such as LINQ and delegates, and where we can best apply them to the AutoCAD API. My hope is that you will walk away with some basic techniques that will make your code more testable, extensible, maintainable, scalable, and reusable.

Learning Objectives

At the end of this class, you will be able to:

- Use delegates to reduce duplicate code
- Explain how LINQ can be used with the AutoCAD API
- Apply abstraction and dependency injection into your C# code
- Use WPF and the Model-View-ViewModel design pattern

About the Speaker

Scott is a senior software engineer for the Infrastructure service line at Woolpert, Inc. He specializes in custom database applications that use Autodesk® software in the AEC, FM and GIS industries. He has more than 30 years of programming experience, and has been integrating databases with AutoCAD® software ever since it was possible. He is the author of AutoCAD Database Connectivity from Autodesk Press, as well as several articles. Scott has attended every AU, and has been a speaker since 1996. He also served two two-year terms on the AUGI® Board of Directors.

scott.mcfarlane@woolpert.com

Introduction

My name is Scott McFarlane. If you're like me, your formal training or college degree was focused on some design-related discipline (I am a degreed architect). You probably also had a strong interest in computer programming, but no formal training. Sure, I took my share of computer programming courses in college with very useful programming languages such as Fortran and Assembler. But generally I consider myself self-taught when it comes to software development. I started using LISP within the first month I ever started using AutoCAD. From there I jumped on each AutoCAD API as it was added. Today, I'm pretty much exclusively a C#.NET developer when it comes to AutoCAD.

In recent years, I have become a "student" of software engineering, software design patterns, and best practices. These practices are the kinds of things I've been doing (in some form or another) for many years, yet these days they are formalized, studied and documented. Let me stress that I will forever be a "student" of these practices – continuously learning and adapting my software design style as my knowledge grows. In this class, while I try to offer a "better way" to solve certain problems, there is likely an "even better way" that I haven't explored. I always welcome discussion and debate on these subjects – so please feel free to ask questions during the class, track me down at the conference, or contact me after the conference is over.

The Importance of High Quality Code

Have you ever encountered an AutoCAD drawing (presumably from someone else!) that was a complete mess? No coherent layer naming, objects on the wrong layer, endpoints not snapped together, blocks exploded, text objects used where block attributes should be, etc. Drawings like this may plot just fine, but working with them inside of AutoCAD can be very difficult. Even more difficult (if not impossible) is trying to work with a messy drawing from a program!

Well, the same scrutiny can (and should) be applied to source code. Messy code can be hard to follow, difficult to maintain, and is prone to defects. Clean, well-structured, high-quality code is code that is not only easy to read and maintain, but is more testable, extensible, scalable, and reusable.

Our Goal: Efficiency

This may be stating the obvious, but as coders, our #1 goal is efficiency. By "efficiency" I don't mean "quick and dirty". We need to be efficient, but at the same time produce clean, high-quality code. Some may see these two goals to be contradictory: How can I be efficient when so much time is spent producing clean code? I have come to realize that attention to quality up-front is like an investment – it will lead to saved time down the road. We have developed good habits to produce quality AutoCAD drawings because we know that this will ultimately save us time down the road. We need to develop similar habits in our software development.

The most obvious way to be efficient in coding is to find ways to generalize common tasks and develop your own toolbox of reusable components that you can share from one project to the next.

Using Delegates to Reduce Duplicate Code

Let's start with one of the most common coding sequences in AutoCAD programming: working with the entities in the current drawing. In this section we will leverage a C# feature called delegates to separate the "boiler plate" code from the code that does the real work.

Consider the following code that finds all the circles in the drawing with a radius less than 1.0 and changes their color to red.

```
[CommandMethod("ENTCOLOR1")]
public void ChangeEntityColor1()
{
    // Get the various active objects
    Document document = Application.DocumentManager.MdiActiveDocument;
    Database database = document.Database;

    // Create a new transaction
    Transaction tr = database.TransactionManager.StartTransaction();

    using (tr)
    {
        // Get the block table for the current database
        var blockTable =
            (BlockTable) tr.GetObject(
                database.BlockTableId, OpenMode.ForRead);

        // Get the model space block table record
        var modelSpace =
            (BlockTableRecord) tr.GetObject(
                blockTable[BlockTableRecord.ModelSpace],
                OpenMode.ForRead);

        RXClass circleClass = RXObject.GetClass(typeof (Circle));

        // Loop through the entities in model space
        foreach (ObjectId objectId in modelSpace)
        {
            // Look for circles
            if (objectId.ObjectClass.IsDerivedFrom(circleClass))
            {
                var circle =
                    (Circle) tr.GetObject(
                        objectId, OpenMode.ForRead);

                if (circle.Radius < 1.0)
                {
                    circle.UpgradeOpen();
                    circle.ColorIndex = 1;
                }
            }
        }
        tr.Commit();
    }
}
```

The majority of this code is necessary for virtually any activity your code needs to perform on the entities in a drawing. In this section, we will explore how the use of delegates, along with extension methods, can greatly reduce the duplication of this code in other similar scenarios.

The "Active" Static Class

Before we go any further, I would like to introduce a simple yet very useful static class called Active, which provides quick access to the most commonly used objects in the AutoCAD API.

```
/// <summary>
/// Provides easy access to several "active" objects in the AutoCAD
```

```
/// runtime environment.
/// </summary>
public static class Active
{
    /// <summary>
    /// Returns the active Editor object.
    /// </summary>
    public static Editor Editor
    {
        get { return Document.Editor; }
    }

    /// <summary>
    /// Returns the active Document object.
    /// </summary>
    public static Document Document
    {
        get { return Application.DocumentManager.MdiActiveDocument; }
    }

    /// <summary>
    /// Returns the active Database object.
    /// </summary>
    public static Database Database
    {
        get { return Document.Database; }
    }

    /// <summary>
    /// Sends a string to the command line in the active Editor
    /// </summary>
    /// <param name="message">The message to send.</param>
    public static void WriteMessage(string message)
    {
        Editor.WriteMessage(message);
    }

    /// <summary>
    /// Sends a string to the command line in the active Editor using String.Format.
    /// </summary>
    /// <param name="message">The message containing format specifications.</param>
    /// <param name="parameter">The variables to substitute into the format string.</param>
    public static void WriteMessage(string message, params object[] parameter)
    {
        Editor.WriteMessage(message, parameter);
    }
}
```

So in other parts of your code, for example, you can access the active database using:

```
var db = Active.Database;
```

Let's modify the previous code sample to use our new Active static class.

```
[CommandMethod("ENTCOLOR2")]
public void ChangeEntityColor2()
{
    using (var tr = Active.Database.TransactionManager.StartTransaction())
    {
        // Get the block table for the current database
        var blockTable =
            (BlockTable)tr.GetObject(
                Active.Database.BlockTableId, OpenMode.ForRead);

        // Get the model space block table record
        var modelSpace =
            (BlockTableRecord)tr.GetObject(
                blockTable[BlockTableRecord.ModelSpace],
                OpenMode.ForRead);

        RXClass circleClass = RXObject.GetClass(typeof(Circle));

        // Loop through the entities in model space
        foreach (ObjectId objectId in modelSpace)
```

```
        {
            // Look for circles
            if (objectId.ObjectClass.IsDerivedFrom(circleClass))
            {
                var circle =
                    (Circle)tr.GetObject(
                        objectId, OpenMode.ForRead);

                if (circle.Radius < 1.0)
                {
                    circle.UpgradeOpen();
                    circle.ColorIndex = 1;
                }
            }
        }
        tr.Commit();
    }
}
```

There are a couple of techniques used in this code worth mentioning, but are beyond the scope of this class.

- Use of the **using** keyword, which automatically calls **Dispose()** on the transaction object when it goes out of scope.
- Use of the **RXObject.GetClass** method along with the **ObjectId.ObjectClass** method, which allows you to check for the type of object before you actually open it.
- Use of the **UpgradeOpen** method, which allows you to only open an object for write when you know you need to.

Introducing Delegates

A *delegate* in .NET is an object that points to another method in your application that can be invoked at a later time. Delegates, like any other type of object, can be passed as arguments to other methods. Along with holding a pointer to a method, delegates also define the method *signature*, that is, the arguments (if any) and the return value (if any).

Let's look at the outer code block of our change color example, which is the part that is responsible for creating, committing and disposing of the **Transaction** object. This is an example of a code fragment that will be repeated any time you do anything with the drawing database.

```
using (var tr = Active.Database.TransactionManager.StartTransaction())
{
    tr.Commit();
}
```

The real work is performed within the using block (and before the **Commit**). Now consider the following **delegate** declaration. Note the use of the **delegate** keyword.

```
public delegate void TransactionFunc(Transaction tr);
```

This declaration defines a delegate (a method signature) that takes a single **Transaction** argument, and returns void (nothing). Now we can define a useful method that takes a **TransactionFunc** as an argument, and invokes the method within the Transaction block.

```
public void UsingTransaction(TransactionFunc action)
{
    using (var tr = Active.Database.TransactionManager.StartTransaction())
```

```
    {
        // Invoke the method
        action(tr);

        tr.Commit();
    }
}
```

To avoid having to explicitly declare delegates, the .NET **System** namespace includes two delegate classes that use *generics*: **Action**, which defines delegates that have no return value, and **Func**, which defines delegates that have a return value. There are multiple versions of each of these classes which support up to 16 method arguments:

- `Action` – no parameters, no return value.
- `Action<T>` – one parameter, no return value.
- `Action<T1, T2>` – two parameters, no return value.
- `Action<T1, T2, T3>` – three parameters, no return value.
- Etc…

And,

- `Func<TResult>` – no parameters, return value of the specified type.
- `Func<T, TResult>` – one parameter, return value of the specified type.
- `Func<T1, T2, TResult>` – two parameters, return value of the specified type.
- `Func<T1, T2, T3, TResult>` – three parameters, return value of the specified type.
- Etc…

So using the **Action<T>** class, we can replace the **TransactionFunc** delegate with its equivalent **Action<Transaction>** and then rewrite our **UsingTransaction** method as follows:

```
public void UsingTransaction(Action<Transaction> action)
{
    using (var tr = Active.Database.TransactionManager.StartTransaction())
    {
        // Invoke the method
        action(tr);

        tr.Commit();
    }
}
```

To use this method, we first take the guts of our code, and put it into a method that has the correct signature:

```
public void ChangeSmallCirclesToRed(Transaction tr)
{
    // Get the block table for the current database
    var blockTable =
        (BlockTable)tr.GetObject(
            Active.Database.BlockTableId, OpenMode.ForRead);

    // Get the model space block table record
    var modelSpace =
        (BlockTableRecord)tr.GetObject(
            blockTable[BlockTableRecord.ModelSpace],
            OpenMode.ForRead);

    RXClass circleClass = RXObject.GetClass(typeof(Circle));
```

```
    // Loop through the entities in model space
    foreach (ObjectId objectId in modelSpace)
    {
        // Look for circles
        if (objectId.ObjectClass.IsDerivedFrom(circleClass))
        {
            var circle =
                (Circle)tr.GetObject(
                    objectId, OpenMode.ForRead);

            if (circle.Radius < 1.0)
            {
                circle.UpgradeOpen();
                circle.ColorIndex = 1;
            }
        }
    }
}
```

Now we can define our command method as follows:

```
[CommandMethod("ENTCOLOR3")]
public void ChangeEntityColorWithDelegate()
{
    UsingTransaction(ChangeSmallCirclesToRed);
}
```

So far, we haven't really saved ourselves many keystrokes with this example use of delegates since all it does is remove the transaction-related code, which really isn't that much code. So let's build on our **UsingTransaction** helper method idea, and strip out some of the boiler plate code related to obtaining the model space block table record.

Consider the following **UsingModelSpace** helper method:

```
public void UsingModelSpace(Action<Transaction, BlockTableRecord> action)
{
    using (var tr = Active.Database.TransactionManager.StartTransaction())
    {
        // Get the block table for the current database
        var blockTable =
            (BlockTable)tr.GetObject(
                Active.Database.BlockTableId, OpenMode.ForRead);

        // Get the model space block table record
        var modelSpace =
            (BlockTableRecord)tr.GetObject(
                blockTable[BlockTableRecord.ModelSpace],
                OpenMode.ForRead);

        // Invoke the method
        action(tr, modelSpace);

        tr.Commit();
    }
}
```

This method allows us to reduce the "circle color change" code to the following method:

```
public void ChangeSmallCirclesToRed2(Transaction tr, BlockTableRecord modelSpace)
{
    RXClass circleClass = RXObject.GetClass(typeof(Circle));

    // Loop through the entities in model space
    foreach (ObjectId objectId in modelSpace)
    {
        // Look for circles
        if (objectId.ObjectClass.IsDerivedFrom(circleClass))
        {
            var circle =
```

```
                (Circle)tr.GetObject(
                    objectId, OpenMode.ForRead);

            if (circle.Radius < 1.0)
            {
                circle.UpgradeOpen();
                circle.ColorIndex = 1;
            }
        }
    }
}
```

And our command method now looks like this:

```
[CommandMethod("ENTCOLOR4")]
public void ChangeEntityColorWithDelegate2()
{
    UsingModelSpace(ChangeSmallCirclesToRed2);
}
```

Now we are starting to save some coding keystrokes, and reduce duplicate code in our applications. Let's take it one step further, and consider the common need to iterate model space and perform some action on all entities of a certain type. Our next helper method will do this, using generics to define the type of object we are looking for.

Here is the code:

```
public void ForEach<T>(Action<T> action) where T: Entity
{
    using (var tr = Active.Database.TransactionManager.StartTransaction())
    {
        // Get the block table for the current database
        var blockTable =
            (BlockTable)tr.GetObject(
                Active.Database.BlockTableId, OpenMode.ForRead);

        // Get the model space block table record
        var modelSpace =
            (BlockTableRecord)tr.GetObject(
                blockTable[BlockTableRecord.ModelSpace],
                OpenMode.ForRead);

        RXClass theClass = RXObject.GetClass(typeof(T));

        // Loop through the entities in model space
        foreach (ObjectId objectId in modelSpace)
        {
            // Look for entities of the correct type
            if (objectId.ObjectClass.IsDerivedFrom(theClass))
            {
                var entity =
                    (T)tr.GetObject(
                        objectId, OpenMode.ForRead);

                action(entity);
            }
        }
        tr.Commit();
    }
}
```

Now our delegate and command methods are greatly simplified:

```
public void ProcessCircle(Circle circle)
{
    if (circle.Radius < 1.0)
    {
        circle.UpgradeOpen();
        circle.ColorIndex = 1;
    }
```

```
}

[CommandMethod("ENTCOLOR")]
public void ChangeEntityColorWithForEach()
{
    ForEach<Circle>(ProcessCircle);
}
```

We can also avoid defining the actual delegate method by using an *anonymous method*. Anonymous methods allow you to define the body of a delegate in-line where the delegate object is expected. So the code for the command method would change to:

```
ForEach(
    delegate(Circle circle)
    {
        if (circle.Radius < 1.0)
        {
            circle.UpgradeOpen();
            circle.ColorIndex = 1;
        }
    });
```

Or, we can use lambda expression syntax like so:

```
ForEach<Circle>(
    circle =>
        {
            if (circle.Radius < 1.0)
            {
                circle.UpgradeOpen();
                circle.ColorIndex = 1;
            }
        });
```

Now that we have this handy ForEach method, we can, for example, print the average length of all the lines in the drawing with just a few lines of code:

```
[CommandMethod("AVGLEN")]
public void AverageLength()
{
    var lengths = new List<double>();
    ForEach<Line>(line => lengths.Add(line.Length));
    Active.WriteMessage("\nThe average length is {0}.", lengths.Average());
}
```

One thing that I don't like about our **ForEach** method is that it always uses the "current" database. It would be better, and more useful, if we could pass it the database we want to work with. Even better, we could make it an *extension method* of the **Database** class. Extension methods allow you to define methods that act just like native methods of a class. Extension methods are defined as static methods of a static class. The static class itself is never actually used. It just acts as a container for the static extension methods.

The signature of an extension method always includes at least one argument, which is of the type to which you want the method to apply. You also prefix that argument with the **this** keyword. Here is our **ForEach** method, implemented as an extension method:

```
public static class ExtensionMethods
{
    public static void ForEach<T>(this Database database, Action<T> action) where T : Entity
    {
        using (var tr = database.TransactionManager.StartTransaction())
        {
            // Get the block table for the current database
```

```
            var blockTable =
                (BlockTable)tr.GetObject(
                    database.BlockTableId, OpenMode.ForRead);

            // Get the model space block table record
            var modelSpace =
                (BlockTableRecord)tr.GetObject(
                    blockTable[BlockTableRecord.ModelSpace],
                    OpenMode.ForRead);

            RXClass theClass = RXObject.GetClass(typeof(T));

            // Loop through the entities in model space
            foreach (ObjectId objectId in modelSpace)
            {
                // Look for entities of the correct type
                if (objectId.ObjectClass.IsDerivedFrom(theClass))
                {
                    var entity =
                        (T)tr.GetObject(
                            objectId, OpenMode.ForRead);

                    action(entity);
                }
            }
            tr.Commit();
        }
    }
}
```

Now our “circle color change” code for the current database can be:

```
Active.Database.ForEach<Circle>(
    circle =>
    {
        if (circle.Radius < 1.0)
        {
            circle.UpgradeOpen();
            circle.ColorIndex = 1;
        }
    });
```

Using LINQ with the AutoCAD API

LINQ, or Language Integrated Query, is another very powerful language feature of .NET that can be leveraged in AutoCAD programs. At its core, LINQ is a set of extension methods tied to the `IEnumerable` and `IEnumerable<T>` types. It also includes a series of keywords that allow you to write SQL-like queries against enumerable types.

Here is a very simple LINQ example:

```
IEnumerable<int> evenNumbers = from i in Enumerable.Range(0, 100)
                                       where i%2 == 0
                                       select i;
```

The above uses the keyword syntax. This can also be written as a method chain:

```
IEnumerable<int> evenNumbers = Enumerable.Range(0, 100).Where(i => i%2 == 0);
```

LINQ is very powerful, and covering all aspects of it is beyond the scope of this class. The focus here will be to demonstrate how LINQ can be used in conjunction with the AutoCAD API. LINQ is all about querying collections of data, and the AutoCAD database (i.e. the drawing) is full of collections of data from symbol tables (layers, blocks, etc.) to entity collections and selection

sets. Naturally, all of these “tables” in the AutoCAD database are exposed in the API through objects that implement `IEnumerable`.

To understand how the LINQ methods work, it is important to understand the distinction between an `IEnumerable` and other “collection” types. Consider the following declarations:

```
IEnumerable<int> a;
List<int> b;
int[] c;
```

What these three types have in common is that they all implement `IEnumerable`, which means that they can all be queried using LINQ. The key distinction between the `IEnumerable` and the other two types is that `List<int>` and `int[]` are themselves containers of data, where `IEnumerable<int>` is separate from its data. `IEnumerable` is an interface that says nothing more than “this object can be enumerated.” It contains a single method, `GetEnumerator()` that returns an `IEnumerator`, which is defined as follows:

```
public interface IEnumerator
{
    bool MoveNext();
    void Reset();
    object Current { get; }
}
```

And `IEnumerator<T>` extends `IEnumerator` as follows:

```
public interface IEnumerator<T> : IEnumerator
{
    new T Current { get; }
}
```

The beauty of `IEnumerator` is that it allows you to sequentially iterate over the elements of a collection without exposing the underlying data structure. This is a common design pattern that is often referred to as the *iterator pattern*.

`IEnumerable` and `IEnumerator` are an integral part of .NET languages. For example, the `foreach` statement in C# requires that the object being enumerated implements the `IEnumerable` interface.

AutoCAD Database Object Enumerators

One of the most common activities in an AutoCAD program that involves enumeration is looping through a collection of database objects. Some of the classes in the AutoCAD .NET API that implement `IEnumerable` for the purposes of enumerating database objects include:

- `SymbolTable` (base class for all symbol tables)
- `AttributeCollection`
- `BlockTableRecord`
- `ObjectIdCollection`
- `SelectionSet`

So how can we leverage the power of LINQ with these classes? Let's first look at the code presented earlier that finds all the circles in the drawing with a radius less than 1.0 and changes their color to red.

```
[CommandMethod("ENTCOLOR1")]
public void ChangeEntityColor1()
{
    // Get the various active objects
    Document document = Application.DocumentManager.MdiActiveDocument;
    Database database = document.Database;

    // Create a new transaction
    Transaction tr = database.TransactionManager.StartTransaction();

    using (tr)
    {
        // Get the block table for the current database
        var blockTable =
            (BlockTable) tr.GetObject(
                database.BlockTableId, OpenMode.ForRead);

        // Get the model space block table record
        var modelSpace =
            (BlockTableRecord) tr.GetObject(
                blockTable[BlockTableRecord.ModelSpace],
                OpenMode.ForRead);

        RXClass circleClass = RXObject.GetClass(typeof (Circle));

        // Loop through the entities in model space
        foreach (ObjectId objectId in modelSpace)
        {
            // Look for circles
            if (objectId.ObjectClass.IsDerivedFrom(circleClass))
            {
                var circle =
                    (Circle) tr.GetObject(
                        objectId, OpenMode.ForRead);

                if (circle.Radius < 1.0)
                {
                    circle.UpgradeOpen();
                    circle.ColorIndex = 1;
                }
            }
        }
        tr.Commit();
    }
}
```

If you look closely at the `foreach` statement, you'll notice that the object being returned by the enumerator is an `ObjectId`. This means that if we were to convert part of that `foreach` statement into a LINQ expression, it might look like this:

```
IEnumerable<ObjectId> circleIds =
                from ObjectId objectId in modelSpace
                where objectId.ObjectClass.IsDerivedFrom(circleClass)
                select objectId;

foreach (ObjectId objectId in circleIds)
{
    var circle =
        (Circle)tr.GetObject(
            objectId, OpenMode.ForRead);

    if (circle.Radius < 1.0)
    {
        circle.UpgradeOpen();
        circle.ColorIndex = 1;
    }
```

```
}
```

We could take it a step further, and include the call to `GetObject` in the LINQ expression. That way the resulting enumerator will return a `Circle`.

```
IEnumerable<Circle> circles =
           from ObjectId objectId in modelSpace
           where objectId.ObjectClass.IsDerivedFrom(circleClass)
           select (Circle)tr.GetObject(objectId, OpenMode.ForRead);

foreach (var circle in circles)
{
    if (circle.Radius < 1.0)
    {
        circle.UpgradeOpen();
        circle.ColorIndex = 1;
    }
}
```

With some more advanced LINQ, we can actually include the logic that checks the radius:

```
IEnumerable<Circle> circlesToMakeRed =
       from ObjectId objectId in modelSpace
       where objectId.ObjectClass.IsDerivedFrom(circleClass)
       select (Circle)tr.GetObject(objectId, OpenMode.ForRead)
           into circle
           where circle.Radius < 1.0
           select circle;

foreach (var circle in circlesToMakeRed)
{
    circle.UpgradeOpen();
    circle.ColorIndex = 1;
}
```

One of the most important things to understand about LINQ is that these expressions take an existing `IEnumerable`, and typically *return a different* ***`IEnumerable`***. This means that none of the code you see in the LINQ expression is executed until you actually iterate through the resulting `IEnumerable`. In all the examples above, nothing really happens until the `foreach` statement is executed.

Ok. Now you're probably thinking, "The LINQ expression above is complete gibberish, and is much more difficult to read than the original `foreach` loop." You are absolutely right. Let's see if we can make use of LINQ in a way that actually makes sense.

Creating a Better Database Object Enumerable

What we really want is an enumerable for database objects that gives us the actual object, and not the `ObjectId`, and does it in a type-safe way. Then we can write code like this:

```
Active.Database.UsingModelSpace(
    (tr, ms) =>
        {
            var circles = ms.OfType<Circle>(tr)
                .Where(c => c.Radius < 1.0);

            foreach (var circle in circles)
            {
                circle.UpgradeOpen();
                circle.ColorIndex = 1;
            }
        });
```

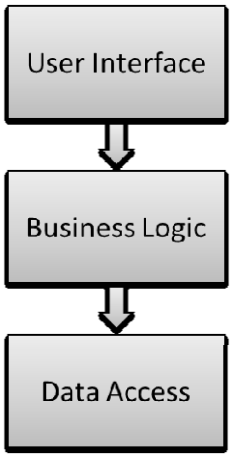
This is actually quite easy:

```
public static IEnumerable<T> OfType<T>(this IEnumerable<ObjectId> enumerable,
        Transaction tr,
        OpenMode openMode)
        where T : DBObject
{
        RXClass rxClass = RXObject.GetClass(typeof(T));

        return
                from ObjectId objectId in enumerable
                where objectId.ObjectClass.IsDerivedFrom(rxClass)
                select (T)tr.GetObject(objectId, openMode);
}
```

Abstraction and Dependency Injection

In a typical application, the components that make up the core business logic will often rely on other components to do their work. For example, if your application uses an external database, you might place your data access code into a separate component. Furthermore, you might separate your presentation tier into its own component. This is a very common practice, and is known as a multi-tiered architecture.



You may also have additional components that perform specific “services” to your application, such as logging, exception handling, configuration, etc. These components all represent *dependencies* upon which other components rely.

Consider the following simple code example, which might represent some business logic code that uses a data tier, and a logging component.

```
public class Example1
{
    public void DoTheWork()
    {
        DataRepository dataRepository = new DataRepository();
        Logger logger = new Logger();

        logger.Log("Getting the data");

        DataSet theData = dataRepository.GetSomeData();

        // Do some work with the data...
```

```
        logger.Log("Done.");
    }
}
```

While it is good that we've separated the data access and logging code into their own components, there are still some issues with this example. First, this method is not only tightly coupled with its dependencies, it is also responsible for their creation. This results in the following issues:

- This code is nearly impossible to reuse because it is so tightly coupled with its dependencies.
- This code would have to be modified if there is a need to replace one of the dependent components with a new implementation.
- This code is impossible to unit test, because it cannot be isolated from its dependencies.

Now examine the following alternative:

```
public class Example2
{
    private readonly IDataRepository _dataRepository;
    private readonly ILogger _logger;

    public Example2(IDataRepository dataRepository, ILogger logger)
    {
        _dataRepository = dataRepository;
        _logger = logger;
    }

    public void DoTheWork()
    {
        _logger.Log("Getting the data");

        DataSet theData = _dataRepository.GetSomeData();

        // Do some work with the data...

        _logger.Log("Done.");
    }
}
```

Here, we've done two key things: 1) We've introduced *abstraction* by creating and developing against *interfaces* rather than specific concrete implementations, and 2) The class is no longer responsible for the creation of its dependencies – they are *injected* into it via the class constructor.

This illustrates the following key principles and design patterns of software engineering:

- **Separation of Concerns** – This class is now only responsible for the specific job it was designed to do.
- **Abstraction** – By using interfaces, we have established a set of *protocols* by which the components interact, separately from the classes themselves.
- **Inversion of Control** – The class has relinquished control of the creation and initialization of its dependencies.
- **Dependency Injection** – This pattern is based on Inversion of Control, and describes the way in which an object obtains references to its dependencies.

Abstraction in AutoCAD Applications

Depending on the nature of your AutoCAD application, you should consider how you might use abstraction to disconnect all or part of your business logic code from its dependency on the AutoCAD API.

Let's explore a simple example that demonstrates how we might accomplish this. Suppose we've been tasked to create an AutoCAD application that is a very simple quantity take-off application. Its goal is to count the number of inserts of each block (by name) in a drawing, and store the results to a database. If we examine the problem in a very abstract way, we can start by writing the core logic as follows:

```
public class BlockCounter
{
    private readonly IDrawing _drawing;
    private readonly IQuantityTakeoffDatabase _quantityTakeoffDatabase;

    public BlockCounter(IDrawing drawing, IQuantityTakeoffDatabase quantityTakeoffDatabase)
    {
        _drawing = drawing;
        _quantityTakeoffDatabase = quantityTakeoffDatabase;
    }

    public void CountTheBlocks()
    {
        var quantityLineItems = new Dictionary<string, QuantityLineItem>();

        foreach (BlockRef blockRef in _drawing.GetAllBlockRefs())
        {
            if (quantityLineItems.ContainsKey(blockRef.Name))
                quantityLineItems[blockRef.Name].Quantity++;
            else
                quantityLineItems.Add(
                    blockRef.Name,
                    new QuantityLineItem
                        {
                            BlockName = blockRef.Name,
                            Quantity = 1
                        });
        }

        _quantityTakeoffDatabase.StoreQuantities(quantityLineItems.Values);
    }
}

public class BlockRef
{
    public BlockRef(string name)
    {
        Name = name;
    }

    public string Name { get; private set; }
}

public class QuantityLineItem
{
    public string BlockName { get; set; }

    public int Quantity { get; set; }
}

public interface IDrawing
{
    IEnumerable<BlockRef> GetAllBlockRefs();
}

public interface IQuantityTakeoffDatabase
{
    void StoreQuantities(IEnumerable<QuantityLineItem> lineItems);
```

```
}
```

By thinking in abstract terms, we've simplified our dependency on the AutoCAD API into a single interface, `IDrawing` that has a single method, `GetAllBlockRefs()`. After all, the only thing we really need from AutoCAD is a way to get the blocks in a drawing. For each block all we care about is the name, so we define a `BlockRef` class, which is nothing more than a simple data structure that represents a block reference with the properties that we need. We've also abstracted the data access component into a single interface, `IQuantityTakeoffDatabase`. What remains is a `BlockCounter` class that contains the core logic behind our application, and it has no dependency on the AutoCAD API. This class can be compiled, tested and debugged in complete isolation – we don't even need to have AutoCAD on the development computer.

Obviously, in order to make our application actually work inside AutoCAD, we need to *implement* the `IDrawing` interface in a separate component that *does* reference the AutoCAD API.

Taking advantage of our `ForEach` method we defined earlier, our implementation of `IDrawing` becomes very easy to write.

```
public class Drawing : IDrawing
{
    public IEnumerable<BlockRef> GetAllBlockRefs()
    {
        var blocks = new List<BlockRef>();

        Active.Database.ForEach<BlockReference>(
            blockRef => blocks.Add(new BlockRef(blockRef.Name)));

        return blocks;
    }
}
```

Next, we need an implementation of `IQuantityTakeoffDatabase`. For now, we'll create a mock implementation that simply spits out the quantity line items to the AutoCAD command window. Our final implementation of `IQuantityTakeoffDatabase` will reside in its own assembly, and will actually write the data to a database somewhere. The beauty of using abstraction is that we don't have to worry about that right now!

```
public class MockDatabase : IQuantityTakeoffDatabase
{
    public void StoreQuantities(IEnumerable<QuantityLineItem> lineItems)
    {
        foreach (QuantityLineItem lineItem in lineItems)
            Active.WriteMessage("{0}: {1}", lineItem.BlockName, lineItem.Quantity);
    }
}
```

Finally, we need a way to invoke the block counter in AutoCAD. The easiest way to do this is to define an AutoCAD command like so:

```
public static class Commands
{
    [CommandMethod("CTB")]
    public static void CountTheBlocks()
    {
        var counter = new BlockCounter(new Drawing(), new MockDatabase());
```

```
        counter.CountTheBlocks();
    }
}
```

At first glance, separating components and using abstraction to define how the components interact may seem like a lot more work. But in fact, it does not increase the amount of code that you write – it just affects how your code is organized. If you put yourself in the mindset of thinking abstractly about a problem, your code will become more reusable, maintainable, scalable, and testable.