Sharpen Your C# Code for AutoCAD

Scott McFarlane
Woolpert, Inc.

Agenda

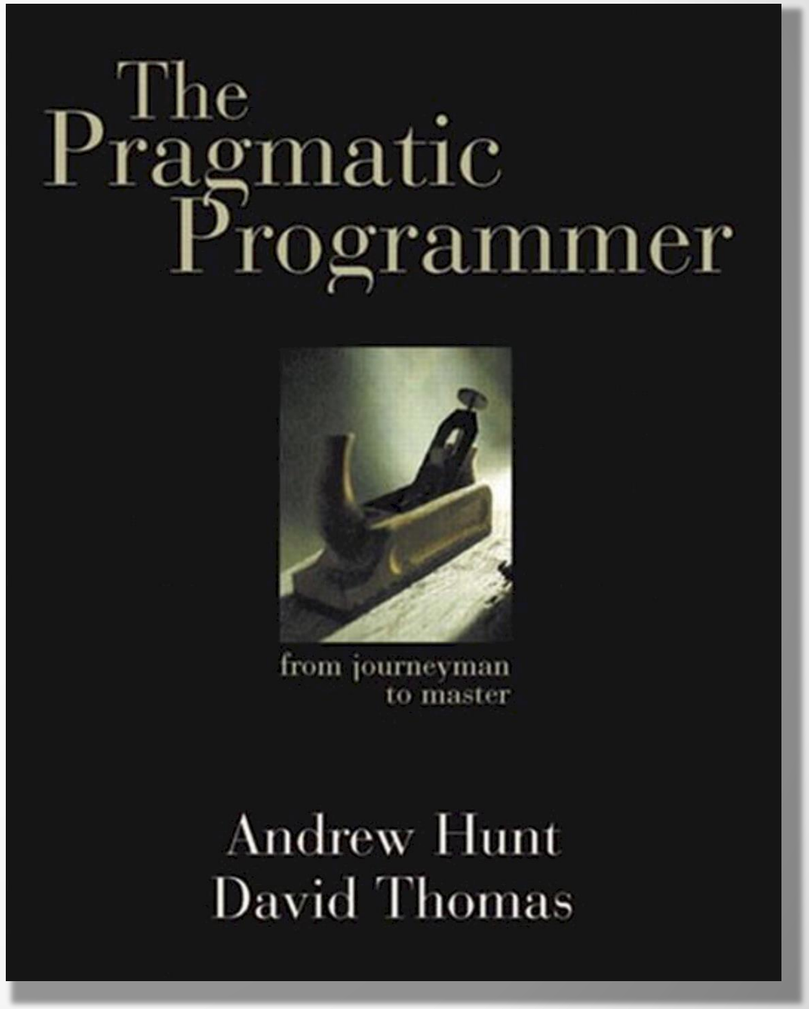
- Using Delegates to Reduce Duplication
- Using LINQ with the AutoCAD API
- Abstraction and Dependency Injection

Topics

- Advanced Language Features
  - Delegates
  - Generics
  - Lambda Expressions
  - Extension Methods
  - LINQ
- Design Patterns
  - Abstraction
  - Dependency Injection
- Tools
  - ReSharper

Our Goal? Eliminate Duplication.

- Don't Repeat Yourself (The DRY Principle)
  - Every piece of knowledge must have a single, unambiguous, authoritative representation within a system.



```
public static void ChangeCircleColor()
{
    // 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);

        // Loop through the entities in model space
        foreach (ObjectId objectId in modelSpace)
        {
            DBObject obj = tr.GetObject(objectId, OpenMode.ForRead);

            // Look for circles
            if (obj is Circle)
            {
                var circle = (Circle) obj;

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

Helper Methods…

```
public static void ChangeCircleColor()
{
    // 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);

        // Loop through the entities in model space
        foreach (ObjectId objectId in modelSpace)
        {
            DBObject obj = tr.GetObject(objectId, OpenMode.ForRead);

            // Look for circles
            if (obj is Circle)
            {
                var circle = (Circle) obj;

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

```
public static class Active
{
    /// <summary>
    /// Gets the active document.
    /// </summary>
    public static Document Document
    {
        get { return Application.DocumentManager.MdiActiveDocument; }
    }

    /// <summary>
    /// Gets the active database.
    /// </summary>
    public static Database Database
    {
        get { return Document.Database; }
    }
}
```

```
public static void ChangeCircleColor()
{
    // Create a new transaction
    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);

        // Loop through the entities in model space
        foreach (ObjectId objectId in modelSpace)
        {
            DBObject obj = tr.GetObject(objectId, OpenMode.ForRead);

            // Look for circles
            if (obj is Circle)
            {
                var circle = (Circle)obj;

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

Delegates…

What is a Delegate?

- A pointer to a function, which can be…
  - Assigned to variables
  - Passed as arguments
  - Invoked through a variable reference

```
public static void ChangeCircleColor()
{
    // Create a new transaction
    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);

        // Loop through the entities in model space
        foreach (ObjectId objectId in modelSpace)
        {
            DBObject obj = tr.GetObject(objectId, OpenMode.ForRead);

            // Look for circles
            if (obj is Circle)
            {
                var circle = (Circle)obj;

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

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

public static void UsingTransaction(TransactionDelegate action)
{
    using (var tr = Active.Database.TransactionManager.StartTransaction())
    {
        try
        {
            action(tr);
            tr.Commit();
        }
        catch (Exception)
        {
            tr.Abort();
            throw;
        }
    }
}
```

Generic Delegate Classes

- **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…

- **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…

```
public static void UsingTransaction(Action<Transaction> action)
{
    using (var tr = Active.Database.TransactionManager.StartTransaction())
    {
        try
        {
            action(tr);
            tr.Commit();
        }
        catch (Exception)
        {
            tr.Abort();
            throw;
        }
    }
}
```

```
public static void DoTheWork(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);

    // Loop through the entities in model space
    foreach (ObjectId objectId in modelSpace)
    {
        DBObject obj = tr.GetObject(objectId, OpenMode.ForRead);

        // Look for circles
        if (obj is Circle)
        {
            var circle = (Circle)obj;

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

public static void ChangeCircleColor()
{
    UsingTransaction(DoTheWork);
}
```

```
public static void ChangeCircleColor()
{
    UsingTransaction(
        delegate(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);

                // Loop through the entities in model space
                foreach (ObjectId objectId in modelSpace)
                {
                    DBObject obj = tr.GetObject(objectId, OpenMode.ForRead);

                    // Look for circles
                    if (obj is Circle)
                    {
                        var circle = (Circle) obj;

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

```
public static void ChangeCircleColor()
{
    UsingTransaction(
        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);

                // Loop through the entities in model space
                foreach (ObjectId objectId in modelSpace)
                {
                    DBObject obj = tr.GetObject(objectId, OpenMode.ForRead);

                    // Look for circles
                    if (obj is Circle)
                    {
                        var circle = (Circle) obj;

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

Extension Methods…

```
public static void UsingTransaction(Action<Transaction> action)
{
    using (var tr = Active.Database.TransactionManager.StartTransaction())
    {
        try
        {
            action(tr);
            tr.Commit();
        }
        catch (Exception)
        {
            tr.Abort();
            throw;
        }
    }
}
```

```
public static void UsingTransaction(this Database database, Action<Transaction> action)
{
    using (var tr = database.TransactionManager.StartTransaction())
    {
        try
        {
            action(tr);
            tr.Commit();
        }
        catch (Exception)
        {
            tr.Abort();
            throw;
        }
    }
}
```

```
public static void ChangeCircleColor()
{
    Active.Database.UsingTransaction(
        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);

                // Loop through the entities in model space
                foreach (ObjectId objectId in modelSpace)
                {
                    DBObject obj = tr.GetObject(objectId, OpenMode.ForRead);

                    // Look for circles
                    if (obj is Circle)
                    {
                        var circle = (Circle) obj;

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

```
public static void UsingBlockTable(this Database database,
    string blockName,
    Action<Transaction, IEnumerable<ObjectId>> action)
{
    database.UsingTransaction(
        tr =>
        {
            // Get the block table
            var blockTable =
                (BlockTable)tr.GetObject(
                    database.BlockTableId, OpenMode.ForRead);

            // Get the block table record
            var tableRecord =
                (BlockTableRecord)tr.GetObject(
                    blockTable[blockName],
                    OpenMode.ForRead);

            // Invoke the method
            action(tr, tableRecord.Cast<ObjectId>());
        });
}

public static void UsingModelSpace(this Database database, Action<Transaction, IEnumerable<ObjectId>> action)
{
    database.UsingBlockTable(BlockTableRecord.ModelSpace, action);
}
```

```
public static void ChangeCircleColor()
{
    Active.Database.UsingModelSpace(
        (tr, modelSpace) =>
            {
                // Loop through the entities in model space
                foreach (ObjectId objectId in modelSpace)
                {
                    DBObject obj = tr.GetObject(objectId, OpenMode.ForRead);

                    // Look for circles
                    if (obj is Circle)
                    {
                        var circle = (Circle) obj;

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

```
public static void ChangeCircleColor()
{
    Active.Database.UsingModelSpace(
        (tr, modelSpace) =>
            {
                RXClass rxClass = RXObject.GetClass(typeof(Circle));

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

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

Generics…

```
public static void ForEach<T>(this Database database, Action<T> action) where T : DBObject
{
    database.UsingModelSpace(
        (tr, modelSpace) =>
        {
            RXClass rxClass = RXObject.GetClass(typeof(T));

            // Loop through the entities in model space
            foreach (ObjectId objectId in modelSpace)
            {
                // Look for circles
                if (objectId.ObjectClass.IsDerivedFrom(rxClass))
                {
                    var obj = (T)tr.GetObject(objectId, OpenMode.ForRead);
                    action(obj);
                }
            }
        });
}
```

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

Using LINQ with the AutoCAD API

Using LINQ with the AutoCAD API

- A very basic LINQ example

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

    foreach (int i in evenNumbers)
        Active.WriteMessage("{0}\n", i);
}
```

- Could also be written as

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

    foreach (int i in evenNumbers)
        Active.WriteMessage("{0}\n", i);
}
```

How Do the Following Declarations Differ?

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

IEnumerable and IEnumerable<T>

- IEnumerable is defined as follows:

```
public interface IEnumerable
{
    IEnumerator GetEnumerator();
}
```

- IEnumerable<T> extends IEnumerable as follows:

```
public interface IEnumerable<T>
{
    IEnumerator<T> GetEnumerator();
}
```

IEnumerator and IEnumerator<T>

- IEnumerator is defined as follows:

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

- IEnumerator<T> extends IEnumerator as follow`s:

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

Some AutoCAD Classes that Implement `IEnumerable`

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

The `Current` property of the `IEnumerator` returns an `ObjectId`.

```
public static void ChangeCircleColor()
{
    Active.Database.UsingModelSpace(
        (tr, modelSpace) =>
            {
                RXClass rxClass = RXObject.GetClass(typeof (Circle));

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

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

```
public static void ChangeCircleColor()
{
    Active.Database.UsingModelSpace(
        (tr, modelSpace) =>
            {
                RXClass rxClass = RXObject.GetClass(typeof (Circle));

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

```
public static void ChangeCircleColor()
{
    Active.Database.UsingModelSpace(
        (tr, modelSpace) =>
            {
                RXClass rxClass = RXObject.GetClass(typeof (Circle));

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

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

```
public static void ChangeCircleColor()
{
    Active.Database.UsingModelSpace(
        (tr, modelSpace) =>
            {
                foreach (var circle in modelSpace.OfType<Circle>(tr))
                {
                    if (circle.Radius < 1.0)
                    {
                        circle.UpgradeOpen();
                        circle.ColorIndex = 1;
                    }
                }
            });
}

public static IEnumerable<T> OfType<T>(this IEnumerable<ObjectId> enumerable, Transaction tr) 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.ForRead);
}
```

```
public static void ChangeCircleColor()
{
    Active.Database.UsingModelSpace(
        (tr, modelSpace) =>
            {
                var circlesToMakeRed =
                    modelSpace
                        .OfType<Circle>(tr)
                        .Where(c => c.Radius < 1.0)
                        .UpgradeOpen();

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

```
public static IEnumerable<T> UpgradeOpen<T>(this IEnumerable<T> enumerable) where T : DBObject
{
    return from obj in enumerable select UpgradeOpen(obj);
}

public static T UpgradeOpen<T>(T obj) where T : DBObject
{
    obj.UpgradeOpen();
    return obj;
}
```

```
public static ObjectId Create<T>(this BlockTableRecord blockTableRecord, Transaction transaction, Action<T> action)
    where T : Entity, new()
{
    var obj = new T();
    obj.SetDatabaseDefaults();
    action(obj);
    var objectId = blockTableRecord.AppendEntity(obj);
    transaction.AddNewlyCreatedDBObject(obj, true);
    return objectId;
}
```

```
public static ObjectId Create<T>(this Database database, Action<T> action)
    where T : Entity, new()
{
    ObjectId objectId = ObjectId.Null;

    database.UsingModelSpace(
        (tr, modelSpace) =>
            {
                objectId = modelSpace.Create(tr, action);
            });

    return objectId;
}
```

```
var id = database.Create<Circle>(circle => circle.Radius = 0.9);
```

Abstraction and Dependency Injection

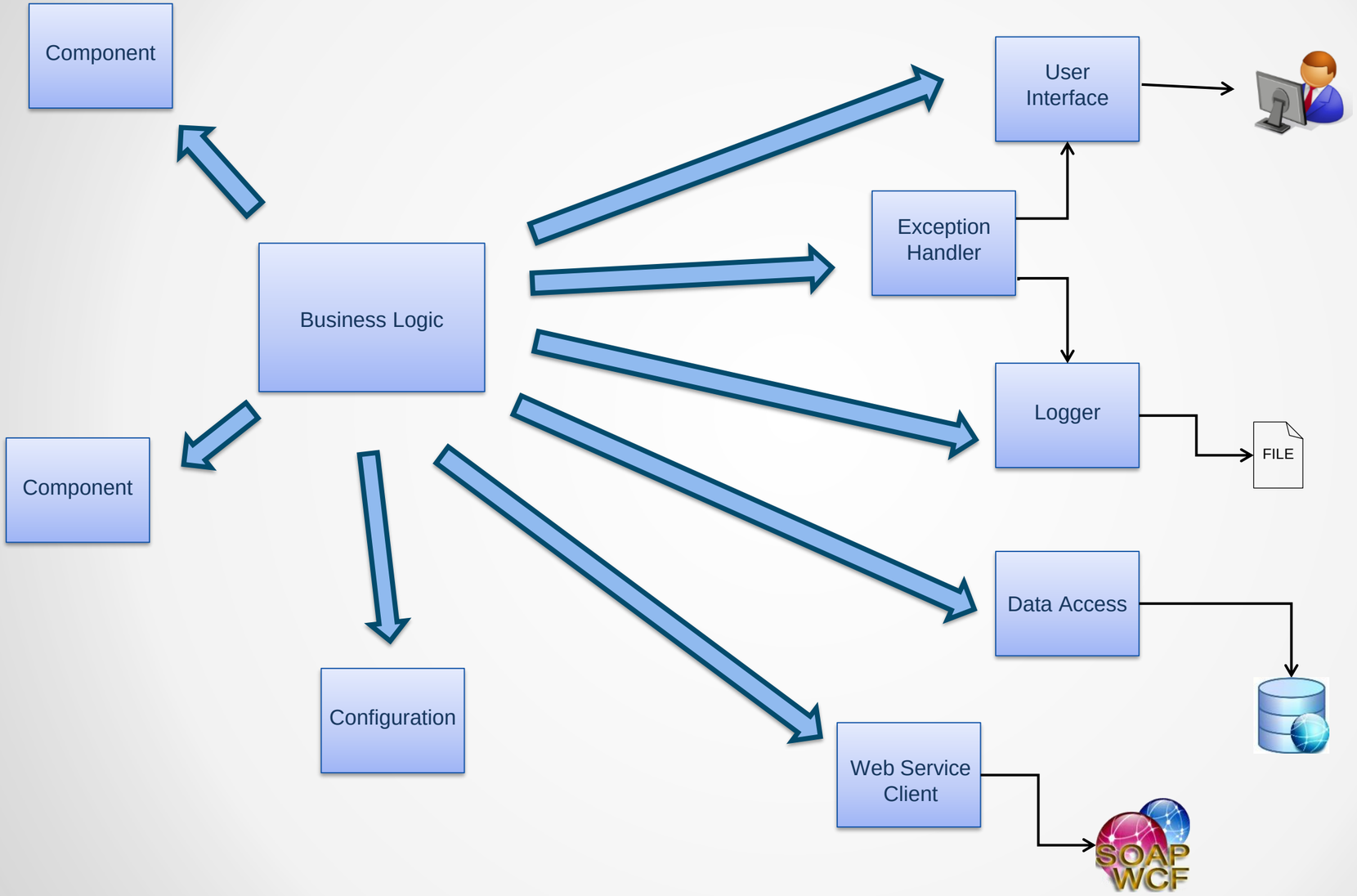
Why is Abstraction Important?

- The Problem
  - Classes often contain dependencies on other classes to do their work.
  - If a class makes assumptions about how its dependent classes are implemented, the class becomes difficult to reuse in combination with other components.
  - Such classes are also very difficult to unit test because they cannot be isolated from their dependencies.

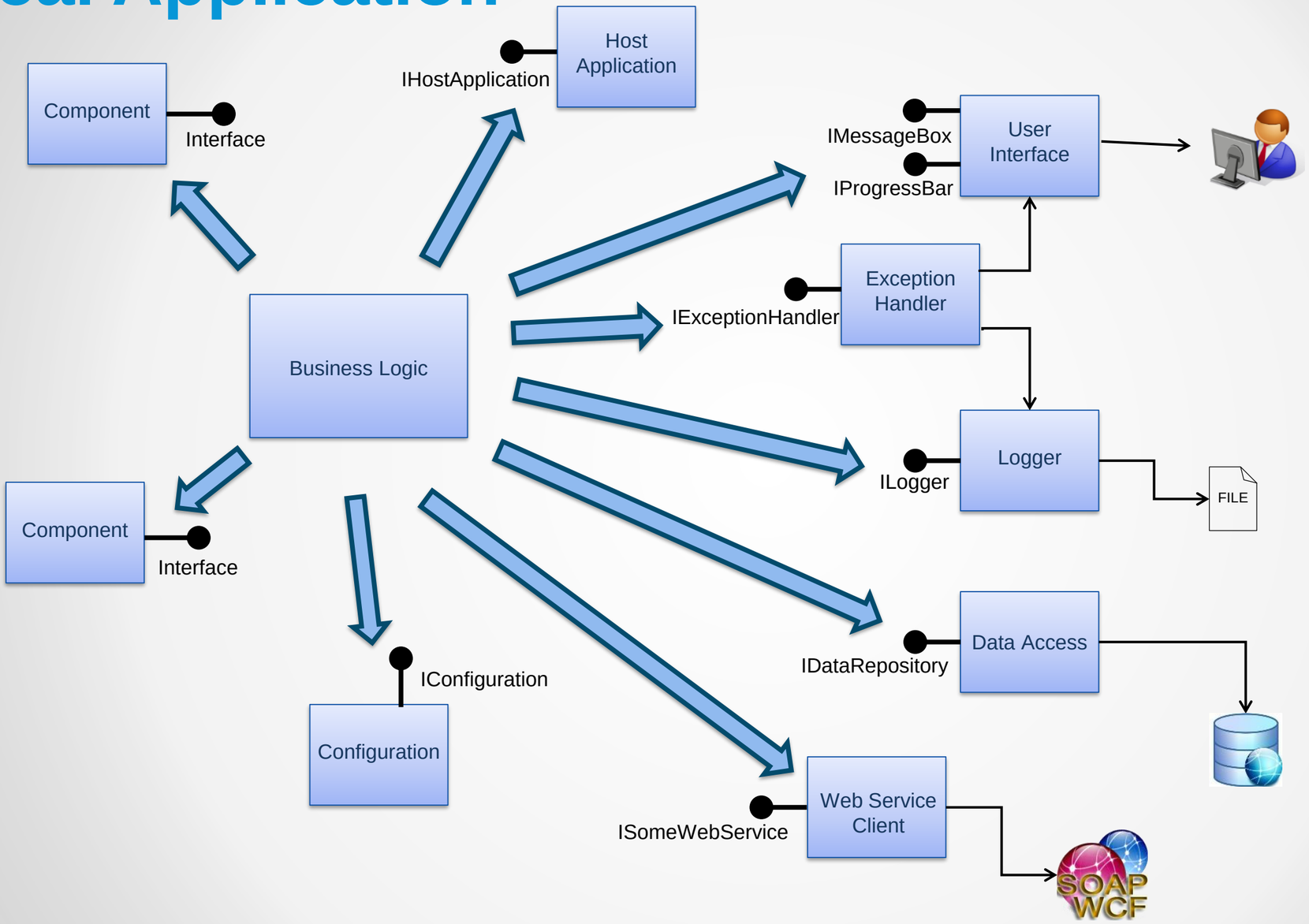
Why is Abstraction Important?

- The Solution
  - Establish a common set of **protocols** by which classes interact, separately from the classes themselves.
  - Promotes the use of software design patterns (particularly *dependency injection*) that result in code that is more testable, extensible, maintainable, scalable, and reusable.

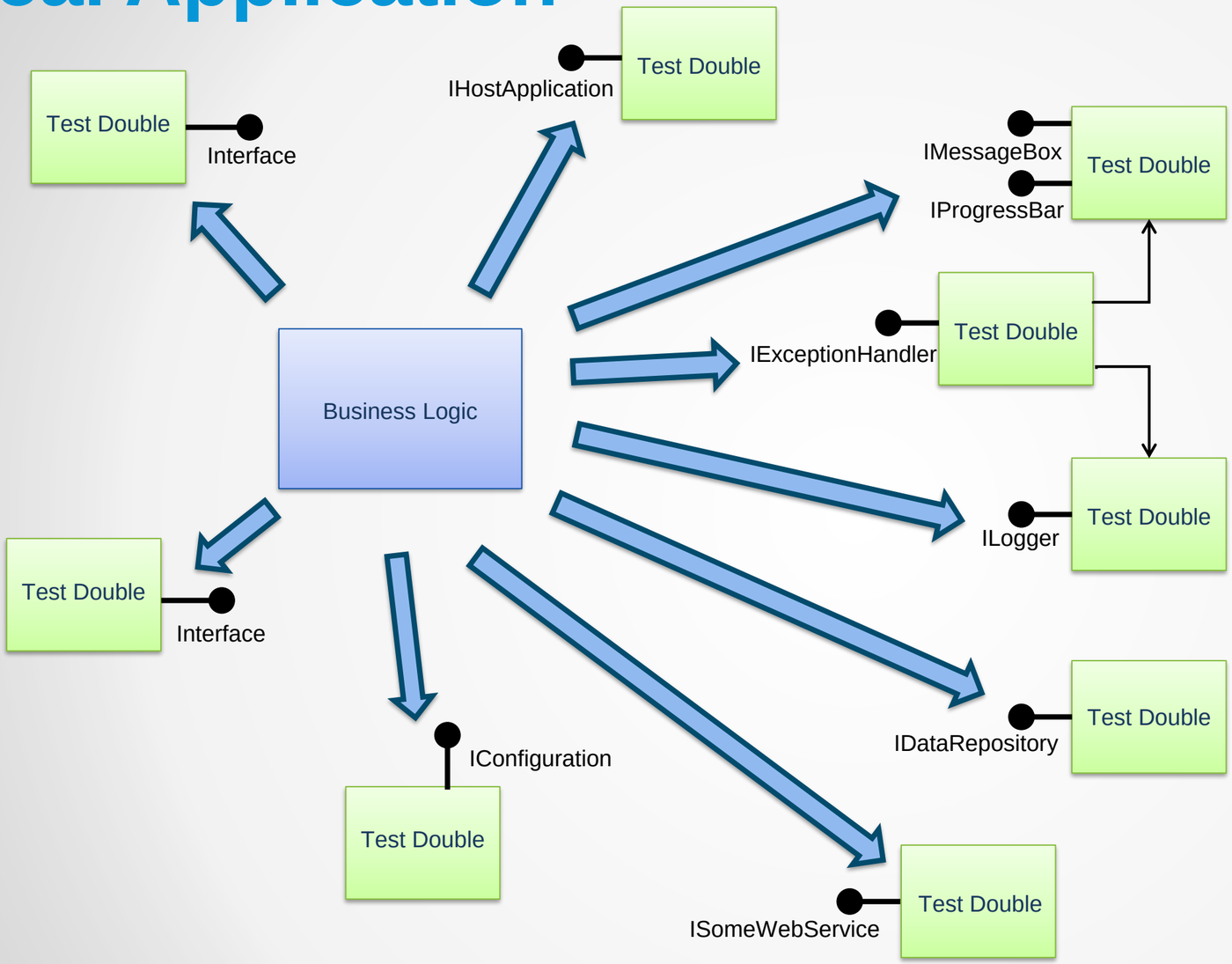
Typical Application



Typical Application



Typical Application



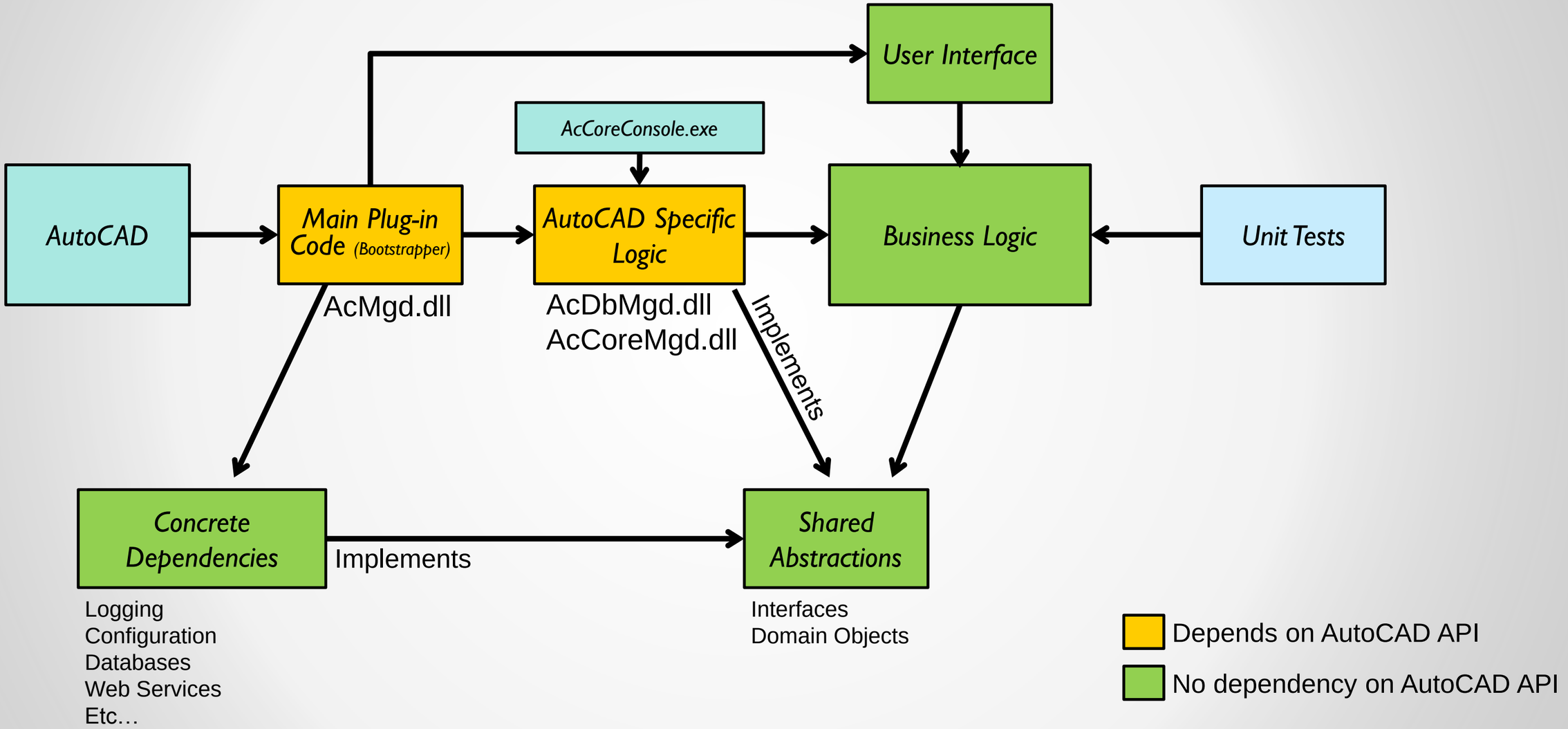
```
/// <summary>
/// This example demonstrates programming against concrete classes.
/// </summary>
public class Example1
{
    /// <summary>
    /// Does the work.
    /// </summary>
    public void DoTheWork()
    {
        var dataRepository = new DataRepository();
        dataRepository.Connect("Data Source=(local);Initial Catalog=MyDatabase;");

        var logger = new Logger();
        logger.Initialize("log.txt");

        logger.Log("Getting the data");

        DataSet theData = dataRepository.GetSomeData();

        // Do some work with the data...

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

```
/// <summary>
/// This example demonstrates programming against interfaces rather than concrete classes.
/// </summary>
public class Example2
{
    private readonly IDataRepository _dataRepository;
    private readonly ILogger _logger;

    /// <summary>
    /// Initializes a new instance of the <see cref="Example2" /> class using constructor injection.
    /// </summary>
    /// <param name="dataRepository">The data repository.</param>
    /// <param name="logger">The logger.</param>
    public Example2(IDataRepository dataRepository, ILogger logger)
    {
        _dataRepository = dataRepository;
        _logger = logger;
    }

    /// <summary>
    /// Does the work.
    /// </summary>
    public void DoTheWork()
    {
        _logger.Log("Getting the data");

        DataSet theData = _dataRepository.GetSomeData();

        // Do some work with the data...

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

Software Engineering Principles

- **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.

Application Architecture



Example Code