CSC320-A

Module 1

Implementing Component-Based Service-Oriented Architecture

1 Encapsulation

1.1 Implementing a Component

A key feature of object-oriented programming is managing complexity. Building large and complex systems can be daunting as humans have limited capacity when dealing with size and complexity. Complexity can be managed by dividing a complex system into small and simple components that can then be individually designed, constructed and tested. Even though each component may still be moderately complex, it is much more manageable by encapsulating it within a class while exposing a simpler interface to access and integrate with the other components. Encapsulation will also protect the internals from unauthorized access and ensures data integrity within the component. When all the components are completed, it would then be very easy to construct the system from a reliable set of simple to use components. In this chapter, you will learn how to encapsulate the complexity of a common and useful component in a class and provide a very simple interface to access its functionality. Components should usually be implemented around a single entity or functionality. If you follow SOLID Principles of Object-Oriented Programming, Single Reponsibility is the first principle.

Many modern applications record information during execution called as logging and usually used by developers and administrators to debug or monitor the working state of the applications. In this section you will create a class to implement the application logging component. Since logging is a common component that can be used across multiple applications, the class should be implemented in a separate class library.

Library project information

```
Project Name: Symbion
Project Type: Visual C# | Windows | Class Library
Location    : C:\CSC320-A\SRC\
Solution    : Module1
```

Before you add a class to implement logging, add a enumeration that is passed to the logging class to determine the type of message to write to the log. Then add the class that display the log message and other details to *Debug* window during debugging. Note that **Assert** and **WriteLine** methods are conditional and will only be called when a *debug version* of the library is used.

LogType enumeration: Symbion\LogType.cs

```
namespace Symbion {
    public enum LogType {
        Information,
        Warning,
        Error
    }
}
```

Logging component class: Symbion\DebugLogger.cs

```
using System;
using System.Diagnostics;
using System.IO;
using System.Reflection;

namespace Symbion {
    public class DebugLogger {
        private string _source;
        public string Source {
            get { return _source; }
            set { Debug.Assert(value != null,
                    "Source property cannot be null.");
                _source = value;}
            }
        }
        public DebugLogger() {
            Assembly assembly = Assembly.GetEntryAssembly();
            Source = Path.GetFileNameWithoutExtension(assembly.Location);
        }
        public void Write(string message, LogType logType) {
            Debug.WriteLine($"[{Source}] {logType}(\"{message}\")");
        }
    }
}
```

Source property is commonly the name of the application using the class and should not be **null**. Passing **null** to it is considered a bug and will be detected by the **Assert** method during debugging. The constructor ensures that the source is never **null** even when the object is created by automatically retrieving the application name. Use an **Assembly.GetEntryAssembly** method to access application assembly and the name extracted from **Location** property that returns the full path of the assembly. Since we only need the name, you can call **GetFileNameWithoutExtension** method from the **Path** class to extract the name from the path. Add an application assembly project to the solution to test out the logging class using the following information.

Application project information

```
Project Name: Encapsulation1
Project Type: Visual C# | Windows | Console Application
Location    : C:\CSDEV\SRC\Module1
```

Main program to test the logging class: Encapsulation\Program.cs

```
using System;

class Program {
      static void Main() {
        DebugLogger logger = new Symbion.DebugLogger();
        Console.WriteLine(logger.Source);
        logger.Write("This is the 1st message.", LogType.Information);
        logger.Write("This is the 2nd message.", LogType.Warning);
        logger.Write("This is the 3rd message.", LogType.Error);
    }
}
```

Run the application in debug mode and you should see the messages appearing in the *Debug output window*. Once you have tested it, let us further simplify usage of the class by setting default values and add helper methods.

1.2 Default Values & Helper Methods

The common log message type is **LogType.Information** so we will assign this as the default value for **logType** parameter in the **Write** method. Thus there is no more need to pass in the second parameter if you intend to write this type of message and the compiler generates code to pass the second parameter.

Setting the default log type: Symbion\DebugLogger.cs

```
public void Write(string message,
    LogType logType = LogType.Information) {
    Debug.WriteLine(string.Format(LogFormat,
        Source, logType, message));
}
```

Using the default value: Encapsulation1\Program.cs

```
logger.Write("This is the 4th message.");
```

Alternatively you can add additional methods that help you to pass the log type so that the compiler would not have to generate the code each time the **Write** method is called.

Adding helper methods: Symbion\DebugLogger.cs

```
public void Message(string message) { Write(message, LogType.Information); }
public void Warning(string message) { Write(message, LogType.Warning); }
public void Failure(string message) { Write(message, LogType.Error); }
```

Using helper methods: Encapsulation1\Program.cs

```
logger.Message("Operation completed successfully.");
logger.Warning("Operation may not have succeeded.");
logger.Failure("Operation has failed.");
```

Encapsulation also allows you to update, change or expand the contents of the class without affecting the code that uses it as long as the class interface it uses does not change. The application does not need to be re-compiled as the class is distributed in a separate assembly. However, since the code is directly using the class, you cannot replace the class with another without changing the code and re-compiling it. In order for the same code to be able to use different classes at run-time you will first need to abstract the class design from the implementation. You can do this by separating the interface from the implementation through abstraction.

2 Abstraction & Inheritance

Abstraction is where the design of a component is separated from its implementation. There are many advantages for abstraction. Assembly dependency is unidirectional so classes that need bidirectional access to each other have to be compiled within the same assembly. However the design can compiled in one assembly and the code that implements the design or uses the design can be compiled in same or other assembly. One design can have multiple implementations supporting exchangeable components. It allows the possibility of polymorphism where one component can be substituted by another compatible component at run-time to build a dynamic system. Abstraction and polymorphism can be implemented using interfaces or base classes but interface is always used to provide the highest level of abstraction. A class is limited to only one base class but can implement any number of interfaces.

2.1 Interface & Abstraction

You can now add an interface to **Symbion** project named as **Ilogger**. Declare all the property and methods from **DebugLogger** in the interface as shown below. You can mark **DebugLogger** as implementing the interface and you can then update the code to use the object through the interface.

Interface declaration: Symbion\ILogger.cs

```
namespace Symbion {
    public interface ILogger {
        string Source { get; set; }
        void Write(string message, LogType logType = LogType.Information);
        void Warning(string message);
        void Message(string message);
        void Failure(string message);
    }
}
```

Class implementing the interface: DebugLogger.cs

```
public class DebugLogger : ILogger {
```

Using object through interface: Abstraction1\Program.cs

```
ILogger logger = new DebugLogger();
```

The key purpose of abstraction is so that you can implement multiple classes having the same interface. You can then write code that can work with objects from any class through the interface. The classes does not require any relationship with each other except that they implement the same interface. Only three types of members can be declared in an interface; *event, property and method*.

2.2 Base Class & Inheritance

Sometimes different classes may still have same or similar code. It is cumbersome to have to write, maintain and update the same code across multiple classes. To remove redundancy, you can create a separate class to place the content instead. Making this class as the base class will allow the code to be inherited into one or more derived classes. Interface is purely abstract so you cannot inherit any code from an interface only the declaration.

Base class used for inheritance: Symbion\BaseLogger.cs

```
using System.Diagnostics;
using System.Reflection;
using System.IO;

namespace Symbion {
    public abstract class BaseLogger : Ilogger {
        private string _source;
        public virtual string Source {
            get { return _source; }
            set { Debug.Assert(value != null,
                    "Source property cannot be null.");
                    _source = value; }
        }
        public BaseLogger() {
            Assembly assembly = Assembly.GetEntryAssembly();
            Source = Path.GetFileNameWithoutExtension(assembly.Location);
        }
        public abstract void Write(string message, LogType logType = LogType.Information);
        public void Message(string message) { Write(message, LogType.Information); }
        public void Warning(string message) { Write(message, LogType.Warning); }
        public void Failure(string message) { Write(message, LogType.Error); }
    }
}
```

Since you may implement different logging components, you can add a base class for them since most of the code that has been implemented in **DebugLogger** is usable in other loggers as well. Rename DebugLogger to **BaseLogger** to turn into a base class. You can mark private members as **protected** to allow derived classes to access them directly if required but this will defeat the encapsulation concept. Use **virtual** to allow derived classes to override members and for members that cannot be inherited can be marked **abstract**. If the class have at least one abstract member the class must be marked abstract as well. You can now create a new DebugLogger class that only require to override the **Write** method. The other members can be inherited from the base class.

Extending from base abstract class: DebugLogger.cs

```
using System.Diagnostics;

namespace Symbion {
    public class DebugLogger : BaseLogger {
        public override void Write(string message, LogType logType = LogType.Information) {
            Debug.WriteLine($"[{Source}] {logType}(\"{message}\")");
        }
    }
}
```

When you extend a class, you always get full inheritance. That means **DebugLogger** will be fully compatible with **BaseLogger** so it is still possible to use a base class for abstraction as shown below. Since **BaseLogger** also has all the members required for **ILogger** you can also set that it implements the interface. This means that all classes that extend **BaseLogger** will automatically support **ILogger** as well.

Using base class for abstraction

```
BaseLogger logger = new DebugLogger();
```

In this chapter, we have satisfied the 2nd and 3rd principles of SOLID; Open-Closed and Liskov Substitution. The Open-Closed principle values extension over modification. A class should always be open for extension and closed for modification. Extend classes to implement new functionality or override existing functionality to support new use classes instead of modifying an existing class to handle them which will make it more complex and difficult to test and maintain. The Liskov Substitution principle denotes that derived classes should always be substitutable for their base classes. This will be better proven in the next chapter where we implement more substitutable classes.

3 Activation & Polymorphism

Abstraction only allows you to access an existing object. The object has to be created first from the class. However the moment you hard-code the class name in the source code, the object would always be an instance of that class. To create a different type of object would require you to modify and rebuild the code. Polymorphism is where you can substitute an object with a different type of object without having to change and recompile code. To accomplish this in our code, you need to use dynamic object activation to instantiate the object and not use the **new** keyword.

3.1 Object Activation

The easiest way to dynamically create an object is to use the **Activator** class. It has many ways to create an object that do not require hard-coded class names. You can create an object by providing assembly name and type name to the **CreateInstance** method. If the assembly is not loaded, it will be loaded automatically. However the method does not return the real object but an **ObjectHandle** instead. You still need to call an **Unwrap** method to access the object. Since the names are just text strings, they can be stored externally rather than using them literally in the source code. This will allow you to create a different type of object and to load in a different assembly without rebuilding the code.

Dynamic object activation: Activation1\Program.cs

```
ILogger logger = (ILogger)Activator.CreateInstance(
       "Symbion","Symbion.DebugLogger").Unwrap();
```

However do not expect developers to write so much code to instantiate an object thus you need to simplify the usage of the **Activator** by implementing another class. Using one class or object to create objects from another class is nothing new in software development as there is a design pattern for it called the *factory method*.

3.2 Factory Method

You will now implement a factory class to encapsulate the complexity that is required and return an instance of a logging class. The actual assembly name and class name is stored in the application configuration file. However, it should not be compulsory that all applications must provide this information. If it is not provided, the factory would use **DebugLogger** as the default.

To test this add an application configuration file into the application assembly project. Then add a setting where the value contains the type and assembly name and the key to fetch it. The key can be anything and value can be in any format you desire. For our example, we use the full interface name as the key and the value contains the names separated by a semicolon. Even though the file is named **App.config**, when you compile the assembly, it will have the same name as the assembly but extended with the **.config** extension which in this case is **Abstraction1.exe.config**.

Application configuration file: Abstraction1\App.config

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
    <appSettings>
        <add key="Symbion.ILogger" value="Symbion.DebugLogger; Symbion"/>
    </appSettings>
</configuration>
```

Even though only applications have configuration files, it can also be accessed from all assemblies with the **ConfigurationManager**. Reference the **System.Configuration** assembly to use it. You can implement a **static** class so that you don't need to create an object from the factory in order to get a logging object.

Factory method instantiate loggers: Symbion\LoggerFactory.cs

```
using System;
using System.Configuration;

namespace Symbion {
    public static class LoggerFactory {
        public const string DefaultLogger = "Symbion.DebugLogger; Symbion";
        private static readonly string _typeName;
        private static readonly string _assemblyName;
        static LoggerFactory() {
            string key = typeof(ILogger).FullName;
            string value = ConfigurationManager.AppSettings[key];
            if (value == null) value = DefaultLogger;
            string[] fields = value.Split(';');
            _typeName = fields[0].Trim();
            _assemblyName = fields[1].Trim();
        }
        public static ILogger CreateInstance() {
            return (ILogger)Activator.CreateInstance(
                _assemblyName, _typeName).Unwrap();
        }
    }
}
```

Instancing logging object through factory: Activation1\Program.cs

```
ILogger logger = LoggerFactory.CreateInstance();
```

3.3 Polymorphism

To demonstrate polymorphism, you will now need to create additional logging classes. However you can implement them in another assembly. To show that we can easily access new loggers without having to compile the application or the **Symbion** library. All that is needed is to update the configuration file. Create a new class library using the following information.

Library project information

```
Project Name : Symbion.Loggers
Project Type : Visual C# | Windows | Class Library
Location     : C:\CSDEV\SRC\Module1
```

Another implementation of ILogger: FileLogger.cs

```
using System;
using System.IO;
namespace Symbion.Loggers {
    public class FileLogger : BaseLogger {
        private string _filename;
        public override string Source {
            get { return base.Source; }
            set {   base.Source = value;
                _filename = Path.ChangeExtension(value, ".log");
            }
        }
        public override void Write(string message, LogType logType) {
            string text = $"\"{DateTime.UtcNow}\",\"{logType}\",\"{message}\"\r\n";
            File.AppendAllText(_filename, text);
        }
    }
}
```

The above **FileLogger** class writes the log message into a file. The next class named as named **EventLogger** will log messages to the system event log instead. Once this is done, you can add the reference to the **Symbion.Loggers** library in the application so that it is deployed to the application folder.

Another implementation of ILogger: EventLogger.cs

```
using System;
using System.Diagnostics;
namespace Symbion.Loggers {
    public class EventLogger : BaseLogger {
        public override void Write(string message,
            LogType logType = LogType.Information) {
            EventLogEntryType entryType = EventLogEntryType.Information;
            if (logType == LogType.Error) entryType = EventLogEntryType.Error;
            else if (logType == LogType.Warning) entryType = EventLogEntryType.Warning;
            EventLog.WriteEntry(Source, message, entryType);
        }
    }
}
```

Referencing assemblies does not mean that they will be automatically loaded if you do not use the types in those assemblies. You can use the following code to check what assemblies are actually loaded at the start of the application.

Code to check assemblies currently loaded: Symbion\DebugHelper.cs

```
public static class DebugHelper {
    public static void ShowLoadedAssemblies() {
        AppDomain domain = AppDomain.CurrentDomain;
        Assembly[] assemblies = domain.GetAssemblies();
        foreach (Assembly assembly in assemblies)
            Console.WriteLine(assembly.FullName);
    }
}
```

Testing Polymorphism: Polymorphism1\Program.cs

```
DebugHelper.ShowLoadedAssemblies();
ILogger logger = LoggerFactory.CreateInstance();
DebugHelper.ShowLoadedAssemblies();
        :
```

You can now build the project and then modify the application configuration file in the application output folder to use a different logging class. There is no need to re-build the solution each time the user wants to change the logger.

4 Advanced Object Access

4.1 Base Classes & Interfaces

If you write code for a specific type, then that code would only work for that type and possibly derived types but would not work for other unrelated types. You would then have to write separate code for each type even though they have the same methods and properties.

Method that works for DebugLogger but not other loggers: Compatibility1

```
static void Log1(DebugLogger logger) {
    logger.Message("Hello!");
    logger.Message("Goodbye!");
}
```

Calling the above method

```
Log1(new DebugLogger());
```

If you target the base class, then the same code can work with all types that derive directly or indirectly from the base class. However any class that only implements the **ILogger** interface but does not inherit from the base class will not be supported.

Method that works for all types derived from BaseLogger

```
static void Log2(BaseLogger logger) {
    logger.Message("Hello!");
    logger.Message("Goodbye!");
}
```

Calling the above method

```
Log2(new DebugLogger());
Log2(new FileLogger());
```

If you target an interface, it then guarantees that all types implementing the interface will be compatible to the code regardless of the base class. This allows the same code to be applied to objects from many different types including future types.

Method that works with any type that implements ILogger

```
static void Log3(ILogger logger) {
    logger.Message("Hello!");
    logger.Message("Goodbye!");
}
```

To make it easier to call these methods that implemented to work with a certain type of object, you can turn them into extension methods. There are 4 requirements to be an extension method; the type that contains the method must be **static**, the method must be **static**, it must accept at least one parameter, the first parameter is marked with **this** keyword. Basically you can replace the class name with the first parameter when you called the method. You don't even need to know the class name as long as the class namespace is imported.

Extension method for ILogger

```
static class Program {
    static void Log4(this ILogger logger) {
        logger.Message("Hello!");
        logger.Message("Goodbye!");
    }
}
```

Calling extension methods

```
new DebugLogger().Log4();
new FileLogger().Log4();
```

4.2 Object Compatibility

Compatibility can be checked at compilation time or runtime. When the exact type of class or interface is specified, the compiler will ensure that the object that you pass in is compatible to that type or interface. For example you can only pass objects to the following method where their types implement the **ICloneable** interface. Code trying to pass incompatible objects cannot be compiled.

Method that accepts only ICloneable objects

```
static object Copy(ICloneable obj) {
    return obj.Clone();     // calling Clone method through ICloneable interface
}
```

Passing items to method

```
var ob1 = Copy("Hello!");           // can compile because String type implements ICloneable
var ob2 = Copy(new ArrayList());    // can compile because ArrayList implements ICloneable
var ob3 = Copy(new int[4]));        // can compile because arrays implements ICloneable
var ob4 = Copy(DateTime.Now);       // cannot compile as DateTime does not support ICloneable
```

You can choose to do runtime compatibility check where you allow any kind of object to be passed in and use the **is** operator to verify compatibily. You can then type-cast the object to the base class or interface in order to use the object.

Runtime instead of compile-time checking

```
static object Copy(object obj) {
    if (obj is ICloneable) {
        ICloneable item = (Icloneable)obj;
        return item.Clone();
    }   return null;    // null is default value if cannot clone
}
```

Passing items to method

```
var ob1 = Copy("Hello!");           // can compile and Clone() will be called
var ob2 = Copy(new ArrayList());    // can compile and Clone() will be called
var ob3 = Copy(new int[4]));        // can compile and Clone() will be called
var ob4 = Copy(DateTime.Now);       // can compile but result is null
```

If the purpose of using the **is** operator is so that you can use the object then it would be simpler to use the **as** operator instead for safe-casting. When you do safe-casting if the object is not compatible, there is no error and **null** will be returned. If not null then the object is compatible and you can use it.

Using safe-casting operator

```
static object Copy(object obj) {
    ICloneable item = obj as ICloneable;
    if (item != null) return item.Clone(); return null;
}
```

Shorter version using ? ternary operator

```
static object Copy(object obj) {
    ICloneable item = obj as ICloneable;
    return item != null ? item.Clone() : null;
}
```

Even shorter version ? null test operator

```
static object Copy(object obj) {
    ICloneable item = obj as ICloneable;
    return item?.Clone();
}
```

4.3 Reflection & Dynamic Objects

In C# 3.0 you are able to create instances of anonymous types by just using the **new** keyword without the type name. You can provide a list of property initializers to store information into the anonymous object. The type of object is known in the method that instantiated the object but will not be known outside of the method. In order for other methods to access the content of the object, you will have to use reflection as shown below.

Display properties of any type: Symbion\DebugHelper.cs

```
public static void DisplayProperties(this object obj) {
    PropertyInfo[] props = obj.GetType().GetProperties();
    foreach (PropertyInfo prop in props) Console.WriteLine("{0}={1}",
            prop.Name, prop.GetValue(obj, null));
}
```

Instancing an anonymous type: Anonymous1

```
static void showAccount(object obj) {
    obj.DisplayProperties();
}
```

```
static void main() {
    var account = new { ID = 101, Name = "ABC Trading", Balance = 50000m };
    Console.WriteLine(account.ID);
    Console.WriteLine(account.Name);
    Console.WriteLine(account.Balance);
    showAccount(account);
}
```

In C# 4.0 you can now use dynamic objects. Dynamic objects are much slower to use and there is no intellisense whatsoever. You can assume that the object has a certain feature and use the feature directly in the code without type information. If the object at runtime does not have that feature, only a runtime error will occur which you can catch and handle. The compiler will generate different code when accessing a dynamic object and does not use reflection.

Using dynamic objects

```
static void showAccount(dynamic obj) {
    Console.WriteLine(obj.ID);
    Console.WriteLine(obj.Name);
    Console.WriteLine(obj.Balance);
}
```

Certain methods may be quite complex to implement based on type compatibility and restrictions. However they are very simple to implement when using dynamic objects. The following method can be used to attempt to add anything to anything and return anything. Whether it works or not depends not on the compiler but the actual objects or values that you pass in at runtime.

A dynamic method to add anything to anything and return anything

```
static dynamic Add(dynamic v1, dynamic v2) {
    return v1 + v2;
}
```

Passing different values and objects to a dynamic method

```
Console.WriteLine(Add(99, 66));
Console.WriteLine(Add(1.99, 66.1));
Console.WriteLine(Add('C', 2));
Console.WriteLine(Add("Hello!", "Goodbye!"));
Console.WriteLine(Add(DateTime.Now, new TimeSpan(1, 2, 3, 4)));
```

5 Service-Oriented Architecture

Modular-Design

A large software system is usually constructed by an entire development team rather than a single programmer. The software system make be designed as an integrated set of modules where each module can be constructed by a different team member. It is possible that modules each work independently or they may also have to integrate to one another. It is important that each module is developed as a separate assembly so that you do not need to recompile the entire software system when there are small changes to a few modules. It is more efficient to compile only changed modules.

Resolving Dependencies

Even if modular-design is the best option you would still encounter dependency issues when modules need to communicate with one another because assembly dependency is unidirectional. This means that if assembly A references assembly B, then B will not be able to reference assembly A. This is because you cannot compile A if B has not been compiled first and you cannot compile B as A is not compiled as well. Abstraction can resolve bidirectional relationships but objects still have to be created somewhere and they have to locate each other during runtime before they can communicate with each other through interfaces. IoC Containers and dependency injection is a pattern that allow objects to resolve references to each other so they can communicate.

5.1 Service Identification

Containers are programmed to hold objects. Third-party containers can hold any type of object since they required to adapt to every application. If you are developing your own container, you can specifically identity the kind of objects that your container can hold. If you are building an application based on a service-oriented architecture where each module provides a set of services, you can first identity which of the components are services. You can then program your container to support service objects. You can use an interface to identify a category of objects.

Declaring interface for services: Symbion\IService.cs

```
namespace Symbion {
    public interface IService {
    }
}
```

Sometimes you can have interfaces without any members at all because they are use for identification rather than function. You are of course free to add more members as requirement to be part of the category defined by the interface.

Once the interface has been created, you can then assign it to any type that is part of the category defined by the interface. If you assign an interface to a base class it will be inherited by all derived classes. Interfaces can also be extended, thus assigning it to an interface guarantee that all types that implement the derived interface will also implement the base interface.

Logging is now a specialized kind of service: ILogger.cs

```
namespace Symbion {
    public interface ILogger : IService {
            :
```

All services may need to share some common features or code so it make sense to create a base class to provide partial or full implementation for them so that they can be inherited instead of having to re-implement the same features in all services.

Generic base class for all services: Symbion\BaseService.cs

```
using System;

namespace Symbion {
    public abstract class BaseService : IService {

    }
}
```

You can extend your existing service classes from the base service class to inherit any features implemented for all services existing or added in the future. Anything added to the above class will be automatically inherited the next time the services are used by a loaded application.

Extending from a generic base class: BaseLogger.cs

```
public abstract class BaseLogger : BaseService, ILogger {
            :
```

5.2 Service Container

You can now implement a class that keeps a repository of service objects. An object can be injected into the container based on what kind of services that it provides. Anywhere the service will be required you can fetch the object based on the type of service. It does not matter what is the class and where is implemented. Add a class named **ServiceRepository** to Symbion. Again you can decide whether the container is object-oriented or **static**.

A simple service container: Symbion\ServiceRepository.cs

```
using System;
using System.Collections.Generic;

namespace Symbion {
    public static class ServiceRepository {
        static Dictionary<Type, IService> _services = new Dictionary<Type, IService>();
    }
}
```

The class contain a dictionary for registering services based on their service type. The **Add** method will be used to register the service while the **Get** method is used to locate the correct service based on the service type. The class is static so that you do not need to instantiate the container before using it. The following shows the code to place the logger into the container and to fetch the logger from the container. The wonderful thing about this is that the code can be compiled into different assemblies that do not have to reference each other. They only have to reference an assembly where the interface is declared which can be constructed just for interfacing.

Method to add service object to container

```
public static bool Add(Type serviceType,IService serviceObject) {
    if (!_services.ContainsKey(serviceType)) {
        _services.Add(serviceType, serviceObject);
        return true; }
    return false;
}
```

Method to retrieve service objects from container

```
public static IService Get(Type serviceType) {
    IService serviceObject = null;
    _services.TryGetValue(serviceType, out serviceObject);
    return serviceObject;
}
```

Example code to add a logger into the container: Dependency1\Program.cs

```
ServiceRepository.Add(typeof(ILogger), LoggerFactory.CreateInstance());
```

Example code to fetch logger from the container

```
ILogger logger = (ILogger)ServiceRepository.Get(typeof(ILogger));
if(logger != null) logger.Write("Logger service located!");
```

While previous code works, the use of **typeof** operator and also typecasting required when fetching the object makes the code looked complicated. You can implement the methods as generic so that you can pass the exact type required to the methods. The methods can adapt to the types required and you can embed **typeof** and typecasting operations within the methods.

Generic version of container methods: Symbion\ServiceRepository.cs

```
public static bool Add<TService>(TService serviceObject)
    where TService : IService {
    Type serviceType = typeof(TService);
    if (!_services.ContainsKey(serviceType)) {
        _services.Add(serviceType, serviceObject);
        return true; } return false;
}

public static Tservice Get<TService>() where TService : IService {
    IService serviceObject = null;
    Type serviceType = typeof(TService);
    _services.TryGetValue(serviceType, out serviceObject);
    return (TService)serviceObject;
}
```

Simpler code to add a logger into the container

```
// ServiceRepository.Add<ILogger>(new DebugLogger());
ServiceRepository.Add(LoggerFactory.CreateInstance());
```

Example code to fetch logger from the container

```
ILogger logger = ServiceRepository.Get<ILogger>();
if(logger != null) logger.Write("Logger service located!");
```

The code to use the container is much simpler due to generic methods. Since the **Add** method is **static** and **ServiceRepository** is a static class, you can turn the method into an extension method of **IService**. This means that the method will automatically appear on any object that is a service.

Turning a static method into an extension method

```
public static bool Add<TService>(this IService serviceObject) {
        :
```

Simplest code to add where container class name is not required

```
// new DebugLogger().Add<ILogger>();
LoggerFactory.CreateInstance().Add();    // CreateInstance returns ILogger
```

5.3 Singleton

Sometimes you do not wish to pre-create an object that may or may not be used and if you create the object, you only want to create it once and share that one instance across the entire application. The container can help you accomplish this but you can also accomplish it without a container or together with a container by implementing it as a singleton. For example, most of the time an application may require a logger but we would not pre-create one unless it is used. Following code shows how to create a singleton for the logging component.

Implement a logger singleton: Symbion/BaseLogger.cs

```
private static Ilogger _instance;    // field to assign singleton
public static ILogger Instance {
    //  return existing singleton or instantiate the singleton
    get {   return _instance ?? (_instance = LoggerFactory.CreateInstance()); }
}
```

Integrate singleton and service container

```
public static ILogger Instance {
    get {
        if (_instance == null) {
            _instance = ServiceRepository.Get<ILogger>();
            if (_instance == null) {
                _instance = LoggerFactory.CreateInstance();
                _instance.Add();
            }
        }   return _instance;
    }
}
```

5.4 Authorization

Large scale applications are not built for one user. An enterprise application is built for everyone in an organization. Different users perform different tasks and should have different levels of security clearances so they cannot get access to something that they are not supposed to have. Thus authorization will be a common service. Since this is common feature for applications, they will be implemented in **Symbion**. Add an interface called **IAuthorization** containing the following members.

Role-based security service: Symbion\IAuthorization.cs

```
using System;
using System.Collections.Generic;

namespace Symbion {
    public interface IAuthorization : IService {
        string UserName { get; }
        bool IsInRole(string roleName);
        bool IsInAnyRoles(IEnumerable<string> roleNames);
        bool IsInAllRoles(IEnumerable<string> roleNames);
    }
}
```

.NET applications uses a principal authorization system. However just as we can freely choose a different authorization service, you can also choose the principal system to use. When building Windows applications, you can then choose to use Windows principal. Set the principal policy for current application to **WindowsPrincipal** and it can then be available to every thread through the **CurrentPrincipal** property.

Authorization with principal object: PrincipalAuthorization.cs

```
using System;
using System.Collections.Generic;
using System.Security.Principal;
using System.Threading;
namespace Symbion {
    public class PrincipalAuthorization : BaseService, IAuthorization {
        private IPrincipal _principal;
        public string UserName {
            get { return _principal.Identity.Name; }
        }
        public PrincipalAuthorization() {
            _principal = Thread.CurrentPrincipal;
        }
        public bool IsInRole(string roleName) { return _principal.IsInRole(roleName); }
        public bool IsInAnyRoles(IEnumerable<string> roleNames) {
            foreach (string roleName in roleNames)
                if (_principal.IsInRole(roleName)) return true;
            return false;
        }
        public bool IsInAllRoles(IEnumerable<string> roleNames) {
            foreach (string roleName in roleNames)
                if (!_principal.IsInRole(roleName))  return false;
            return true;
        }
    }
}
```

Adding authorization service into the container: Dependency1\Program.cs

```
AppDomain.CurrentDomain.SetPrincipalPolicy(PrincipalPolicy.WindowsPrincipal);
new PrincipalAuthorization().Add<IAuthorization>();
```

Fetching and using authorization service

```
var auth = ServiceRepository.Get<IAuthorization>();
Console.WriteLine(auth.UserName);
Console.WriteLine(auth.IsInRole("Administrators"));
Console.WriteLine(auth.IsInRole("Banking"));
```

For security concerns, you should also try to use domain roles instead of local roles as a user might be able to obtain administrative privileges to the local computer to add themselves to the required roles but not when on a remote domain server.

6 Modules & Serialization

6.1 Modules

You can think a module as a dynamic component of an application loaded only when necessary. The module can contain a set of services that can be either used internally or exposed externally for integration to other modules. A module may contain views to allow user to interact visually with the module and its services. To allow modules to perform initialization and finalization operations, the interface should expose an **Init** and **Exit** method. You can implement a base class for this interface to provide blank implementations since not every module require implementing the members.

Basic interface for modules: Symbion\IModule.cs

```
namespace Symbion {
    public interface IModule {
        void Init();
        void Exit();
    }
}
```

Base class for modules: Symbion\BaseModule.cs

```
using System;

namespace Symbion {
    public class BaseModule : IModule {
        public virtual void Init() { }
        public virtual void Exit() { }
    }
}
```

6.2 Serialization

Smart client applications commonly check what modules are available and determine which modules is to be loaded depending on the user is running the application. It would not be appropriate to load modules that the user will never use or safe to allow the user to access modules that they are not authorized to use. Each module should have a class used to initialize a module when loaded. Add a **BankingModule** class to **SymBank.Banking** project. Extend **BaseModule** to inherit the default code. We will add more code to this class later.

Basic implementation of a module class: BankingModule.cs

```
using System;
using Symbion;

namespace SymBank.Banking {
    public class BankingModule : BaseModule {
    }
}
```

We can implement a loader to dynamically load modules. However, before this can be done, we need a data model to store information about what modules are available, who are authorized to use them and where are the location of the modules and also the name of the module class. While you can use application configuration file to store simple settings but you have to perform a lot more work to use it for storing custom objects. It will be simpler to implement your data model in code and use serialization to store and load objects. In this example, we will use XML serialization.

You can add a **ModuleItem** class to the Symbion library. The purpose of this class is to store the location and name of the module, and also a list of roles to determine the users using this module. If the user is not part of any of the roles, the module will not be loaded. This is where we can make use of the authorization service that we have implemented previously. You can add a **ModuleList** class to the library. This class is used to store a collection of **ModuleItem** elements. Add a static **Load** method to de-serialize an instance from an XML document by using the **XmlSerializer**. Also provide a method to serialize an instance into an XML document so that it will be possible to create the document by writing code. The XML document can then be edited outside of the application.

Class to store information about a single module: ModuleItem.cs

```
using System;
using System.Collections.Generic;
namespace Symbion {
    public class ModuleItem {
        public string Name { get; set; }
        public string Path { get; set; }
        public List<string> Roles { get; set; }
        public ModuleItem() { Roles = new List<string>(); }
    }
}
```

Class to store a list of module items: ModuleList.cs

```
using System;
using System.Collections.Generic;
namespace Symbion {
    public class ModuleList {
        public List<ModuleItem> Items { get; set; }
        public ModuleList() { Items = new List<ModuleItem>(); }
    }
}
```

Methods to serialize and de-serialize a ModuleList from XML

```
public void Save(string path) {
    XmlSerializer serializer = new XmlSerializer(typeof(ModuleList));
    FileStream stream = new FileStream(
             path, FileMode.Create, FileAccess.Write);
    try { serializer.Serialize(stream, this); }
      finally { stream.Close(); }
}

public static ModuleList Load(string path) {
    if (!File.Exists(path)) return new ModuleList();
    XmlSerializer serializer = new XmlSerializer(typeof(ModuleList));
    FileStream stream = new FileStream(path, FileMode.Open, FileAccess.Read);
    try { return (ModuleList)serializer.Deserialize(stream); }
    finally { stream.Close(); }
}
```

6.3 Serialization Format

If you want to customize the serialization then you need to serialize an example first so that you can see the serialized XML format. Add a console application to the same solution named **Serialize1** and write code to construct the model and call **Save**.

Serializing our object model to XML: Serialization1\Program.cs

```
using System;
using Symbion;
class Program {
    static void Main() {
        ModuleList list = new ModuleList();
        ModuleItem item = new ModuleItem();
        item.Name = "SymBank.Banking.BankingModule";
        item.Path = "SymBank.Banking.dll";
        item.Roles.Add("Administrators");
        item.Roles.Add("Banking");
        list.Items.Add(item);
        list.Save("Modules.xml");
    }
}
```

Output format of the above serialization: Modules.xml

```
<?xml version="1.0"?>
<ModuleList xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <Items>
    <ModuleItem>
      <Name>SymBank.Banking.BankingModule</Name>
      <Path>SymBank.Banking.dll</Path>
      <Roles>
        <string>Administrators</string>
        <string>Banking</string>
      </Roles>
    </ModuleItem>
  </Items>
</ModuleList>
```

The above format is usable but we are going to change the format slightly so it can be edited easier by hand. First change the name of the root element from *ModuleList* to *ModuleCatalog* by attaching an **XmlType** attribute to the class. Also change the name of the *Items* property to *Modules*. Use **XmlElement** and **XmlAttribute** to change the name of properties but since this is a collection property, use the **XmlArray** attribute instead to change the name.

We also prefer that a *ModuleItem* is serialized as *Module*. It would be simpler to edit if we made *Name* and *Path* as XML attributes rather than as elements. This can be done using **XmlAttribute**. Change also the name of each item in the *Roles* collection which is now serialized as *string*. You can change name of serialized items in an array or collection using **XmlArrayItem**. Anything that you do not want to serialized, you can mark it using **XmlIgnore**. By default all public fields and properties will be serialized when using XML serialization.

Changing serialized type and property name: ModuleList.cs

```
[XmlType("ModuleCatalog")]
public class ModuleList {
    [XmlArray("Modules")]public List<ModuleItem> Items { … }
                :
```

Customizing ModuleItem serialization: ModuleItem.cs

```
[XmlType("Module")]
public class ModuleItem {
    [XmlAttribute]public string Name { … }
    [XmlAttribute]public string Path { … }
    [XmlArrayItem("Role")]public List<string> Roles { … }
                :
```

Output of new serialization format

```
<?xml version="1.0"?>
<ModuleCatalog
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <Modules>
        <Module
            Name="SymBank.Banking.BankingModule"
            Path="SymBank.Banking.dll">
            <Roles>
                <Role>Administrators</Role>
                <Role>Banking</Role>
            </Roles>
        </Module>
    </Modules>
</ModuleCatalog>
```

You can now implement a **ModuleLoader** class in Symbion project. The class keeps a collection of **IModule** objects that have been instantiated from the loaded modules. The **Init** method will be used to initialize all modules and the **Exit** method to finalize all modules. The **Load** method is called with the path to the XML file containing the **ModuleList** which will then load in all authorized modules.

ModuleLoader class implementation: Symbion\ModuleLoader.cs

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Reflection;
using System.IO;

namespace Symbion {
    public static class ModuleLoader {
        private static List<IModule> _modules = new List<IModule>();
        public static void Init() {
            foreach (IModule module in _modules) module.Init(); }
        public static void Exit() {
            foreach (IModule module in _modules) module.Exit(); }
        public static void Load(string path) { }
    }
}
```

You can use **Assembly.LoadFrom** method to load in an assembly dynamically from a path. Once an assembly is loaded you can locate a **Type** by name using the **GetType** method or **GetExportedTypes** method to obtain all public types in the assembly.

Implementation of the Load method

```
IAuthorization auth = ServiceRepository.Get<IAuthorization>();
ILogger logger = ServiceRepository.Get<ILogger>();
ModuleList list = ModuleList.Load(path);
foreach (ModuleItem item in list.Items) {
    if (item.Roles.Count > 0 && !auth.IsInAnyRoles(item.Roles)) {
        Debug.WriteLine(string.Format("User not authorized for module {0}.", item.Path));
        continue;
    }
    if (!File.Exists(item.Path)) {
        logger.Failure(string.Format("Cannot locate module {0}.", item.Path));
        continue;
    }
    Assembly assembly = null;
    try { assembly = Assembly.LoadFrom(item.Path); }
    catch (Exception ex) {
        logger.Failure(string.Format(
            "Error '{0}' occurred loading module {1}.", ex.Message, item.Path));
        continue;
    }
    Type moduleType = assembly.GetType(item.Name);
    if (moduleType == null) {
        logger.Failure(string.Format(
            "Cannot find class {0} in module {1}.", item.Name, item.Path));
        continue;
    }
    try {
        _modules.Add((IModule)Activator.CreateInstance(moduleType));
        Debug.WriteLine(string.Format("Module {0} loaded successfully.", item.Path));
    }
    catch (Exception ex) {
        logger.Failure(string.Format(
            "Error '{0}' instancing {1} in module {2}.", ex.Message, item.Name, item.Path));
    }
}
```

You can now make use of the **ModuleLoader** to load in any dynamic modules for an application. All the modules can be initialized and finalized by using the **Init** and **Exit** methods. You can add a new library project to the solution to implement an example module.

Library project information

```
Project Name: SymBank.Banking
Project Type: Visual C# | Windows | Class Library
Location    : C:\CSDEV\SRC\Module1
```

Service Interface: SymBank.Banking\Services\IAccountController.cs

```
public interface IAccountController : IService  { }
```

Service Interface: SymBank.Banking\Services\ITransactionController.cs

```
public interface ItransactionController : IService { }
```

Service class implementation: SymBank.Banking\Controllers\BankingController.cs

```
public class BankingController : IAccountController, ITransactionController { }
```

Example module initializing services: SymBank.Banking\BankingModule.cs

```
namespace SymBank.Banking {
    public class BankingModule : BaseModule {
            public override void Init() {
                var obj = new BankingController();
                obj.Add<IAccountController>();
                obj.Add<ITransactionController>();
            }
    }
}
```

Loading and finalizing modules: Dependency1\Program.cs

```
AppDomain.CurrentDomain.SetPrincipalPolicy(PrincipalPolicy.WindowsPrincipal);
new PrincipalAuthorization().Add<IAuthorization>();
LoggerFactory.CreateInstance().Add();
ModuleLoader.Load("Modules.xml");
ModuleLoader.Init();
ModuleLoader.Exit();
```

Add *Modules.xml* file to the application project. Use properties window to ensure that file is set as **Content** and *Copy To Output Directory* option is set to **Copy if newer**. This ensures the file will be deployed together with the application. As **DebugLogger** is used by default, check the debug output window to see the results. If the user is authorized, the module will be loaded and initialized.

The above module demonstrates the 4th SOLID principle, Interface Segregation where it is up to you to decide whether each interface is implemented by a separate class or a class can choose to implement both interfaces. By segregating the functionality into interfaces, it becomes a flexibility for components to decide what it supports and what it does not.

7 Attributes & Automation

7.1 .NET Attributes

You can use .NET attributes to attach information to program element that includes the assembly, class, structure, interface, delegate, field, event, property and method. Each .NET attribute is an object that is instantiated from a class that is extended from a base **Attribute** class. This will allow anyone to create their own custom attributes. Reflection can be used during runtime to obtain custom attributes from types. When implementing your custom attribute class, the class name must always end with **Attribute**. To limit which program element that your attribute can be attached to and also if the attribute can be attached more than once to a single element, use the **AttributeUsage** attribute on your class. You can decide whether an attribute can only be used once on each element by setting **AllowMultiple** to false. The following is an attribute class named **ServiceAttribute** that can be attached to a class. It is up to you to decide if you need to store data in your attributes. If so you can then create a custom constructor to allow the data to be passed in easily and provide properties to retrieve the data later. You may also allow properties to be set separately rather than passed through the constructor.

Service attribute class: Symbion\ServiceAttribute.cs

```
namespace Symbion {
    [AttributeUsage(AttributeTargets.Class, AllowMultiple = true)]
    public sealed class ServiceAttribute : Attribute {
        private Type _serviceType;
        public Type ServiceType {
            get { return _serviceType; }
            set { _serviceType = value; }
        }
        public ServiceAttribute() { }
        public ServiceAttribute(Type serviceType) {
            _serviceType = serviceType;
        }
    }
}
```

Assigning attribute to class: SymBank.Banking\Controllers\BankingController.cs

```
    [Service(typeof(IAccountController))]
    [Service(typeof(ITransactionController))]
//  [Service(ServiceType = typeof(IAccountController))]
//  [Service(ServiceType = typeof(ITransactionController))]
    public class BankingController : IAccountController, ITransactionController {
    }
```

7.2 Automation

Operations such as creating and injecting objects into a container can be automated through an application framework. Previously we implemented an **ServiceAttribute** that can be attached to any class. Our application framework can specifically look for this type of attribute by examining all types exported from an assembly. You can call **GetCustomAttributes** method on any **Type** to locate specific attribute types. The method returns an array since you may be able to attach multiple instances of an attribute if **AllowMultiple** is **true**. You can determine if inherited attributes is allowed using a second parameter. We will now add a new method to **ServiceRepository** class. This method check all the types in assembly containing the module class to locate all classes that have been attached the attribute. It will then create an instance of that object to be registered as a service by using the service type in the attribute. Change the **Init** method in **ModuleLoader** to inject every module loaded.

Auto-instancing and registration of services: ServiceRepository.cs

```
public static void AddServices(this IModule module) {
    Assembly assembly = module.GetType().Assembly;
    Type[] types = assembly.GetExportedTypes();
    foreach (Type type in types) {
        if (!type.IsClass) continue;
        var attributes = (ServiceAttribute[])
            type.GetCustomAttributes(typeof(ServiceAttribute), false);
        if (attributes.Length == 0) continue;
        var instance = (IService)Activator.CreateInstance(type);
        foreach (var attribute in attributes) {
            Type serviceType = attribute.ServiceType;
            if (!_services.ContainsKey(serviceType))
                _services.Add(serviceType, instance);
        }
    }
}
```

Auto-injection of module services: ModuleLoader.cs

```
public static void Init() {
    foreach (IModule module in _modules) {
        module.Init();
        module.AddServices();
    }
}
```

So now there is no need for a module to have to instantiate and register any services. It will be automatically be done just by attaching the attribute to any service class. With automation, developers can write less code to perform standard operations and having less code usually means less bugs.

8 Dependency Injection

8.1 Constructor Injection

If services can be automatically registered, then services be automatically retrieved from the container as well. This is called as dependency injection. There are different ways to perform dependency injection, one method is to through the constructor. The class that require services will define a constructor that will accept the services that it needs to use. We will then use the container to create the object from this class. The container will then attempt to resolve all the dependencies and then create the object using those dependencies.

Method to perform constructor injection: ServiceRepository.cs

```
public static TInstance CreateInstance<TInstance>() where TInstance : class {
    Type type = typeof(TInstance);
    ConstructorInfo constructor = type.GetConstructors()[0];
    ParameterInfo[] parameters = constructor.GetParameters();
    List<object> values = new List<object>();
    foreach (ParameterInfo parameter in parameters) {
        IService serviceObject = Get(parameter.ParameterType);
        if (serviceObject == null) throw new Exception(
            $"Cannot resolve parameter {parameter.Name}");
        values.Add(serviceObject);
    }
    return (TInstance)constructor.Invoke(values.ToArray());
}
```

To demonstrate this, add the following class with auto-properties for services that we need to use. The constructor will accept references to these services and assign them to the properties.

Class with constructor injection: Dependency1\BankingWorker.cs

```
public class BankingWorker : IService {
    public IAccountController Account {get; set; }
    public ITransactionController Transaction { get; set; }

    public BankingWorker(
        IAccountController account,
        ITransactionController transaction) {
        Account = account;
        Transaction = transaction;
    }
}
```

So all you have to do is use **CreateInstance** from the **ServiceRepository** container to instantiate the object. Examine the properties and you will find that they already contain references to the required services.

Demonstration of constructor injection: Program.cs

```
var worker = ServiceRepository.CreateInstance<BankingWorker>(); worker.Add();
Console.WriteLine(worker.Account);
Console.WriteLine(worker.Transaction);
```

8.2 Property Injection

Another way is to perform property injection where properties can be marked with a specific attribute. The container can then resolve the dependencies by looking for the properties with the attribute and then attempt to fetch the required service from the container and assign to the property.

Attribute for dependency injection: Symbion\InjectAttribute.cs

```
using System;

namespace Symbion {
    [AttributeUsage(AttributeTargets.Property, AllowMultiple = false)]
    public sealed class InjectAttribute : Attribute {
        public InjectAttribute() { }
    }
}
```

Method to resolve property injections: Symbion\ServiceRepository.cs

```
public static void Inject(this object obj) {
    Type type = obj.GetType();
    PropertyInfo[] properties = type.GetProperties();
    foreach (PropertyInfo property in properties) {
        if (property.GetCustomAttribute<InjectAttribute>() != null) {
            IService serviceObject = Get(property.PropertyType);
            if (serviceObject != null) property.SetValue(obj, serviceObject);
        }
    }
}
```

You can now use the above **Inject** attribute to mark each property that requires the reference to a service object. Call **Inject** on the container on the object either in the constructor or after the object has been created. Examine the properties to check that the references to the services has already been resolved.

Properties with Inject attribute: Dependency1\BankingWorker.cs

```
public class BankingWorker : IService {
    [Inject]public IAccountController Account {get; set; }
    [Inject]public ITransactionController Transaction { get; set; }
    public BankingWorker() { this.Inject() }
}
```

Demonstration of property injection: Program.cs

```
var worker = new BankingWorker();
Console.WriteLine(worker.Account);
Console.WriteLine(worker.Transaction);
```

This completes the 5th principle of SOLID; Dependency Inversion. By implementing a Inversion of Controller (IOC) container with Dependency Injection (DI) support, high-level components can depend on low-level components through abstractions as well. This allow all components to be substitutable at all levels of integration.

SOLID Principles

```
(S)ingle-Reponsibility
(O)pen-Closed
(L)Liskov Substitution
(I)Interfae Segregation
(D)Dependency Inversion
```