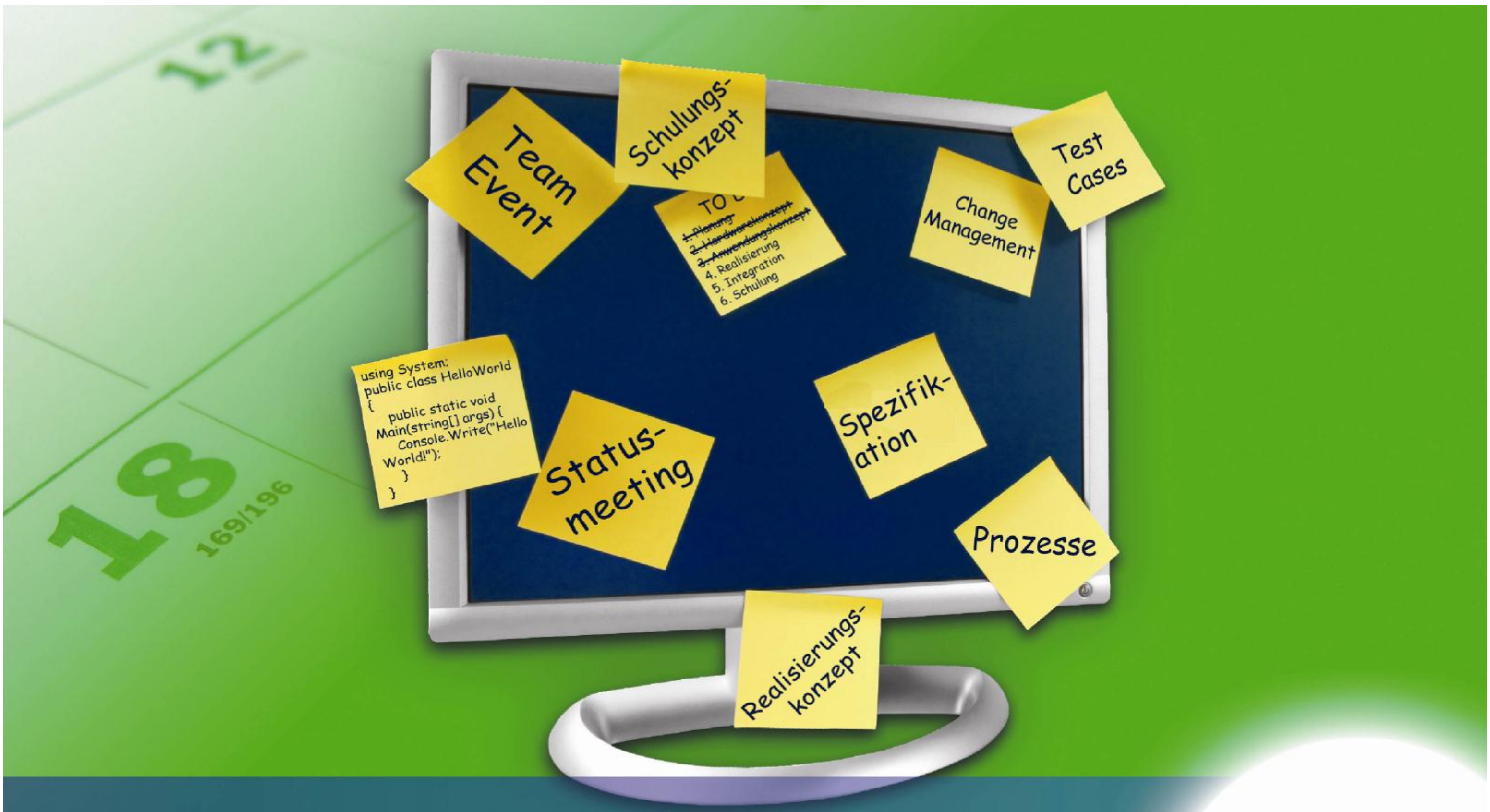

Show Case - Equals

Hands on Labs

rubicon informationstechnologie gmbh

Table of Contents

1 Goal 1

2 Lab 1: Equals without Mixins 1

2.1 Theory: Equals 1
2.2 Exercise 1: Basic Equals Implementation 2
2.2.1 Task 1: Implementation 2
2.2.2 Questions / Excercises 3
2.3 Theory: Reflection 3
2.4 Exercise 2: Using Inheritance 3
2.4.1 Task 1: Implementation 3
2.4.2 Questions / Excercises 5
2.5 Exercise: Utility Implementation 5
2.5.1 Task 1: Implementation 5
2.5.2 Questions / Exercises 6
2.6 Exercise: Strategy Implementation 7
2.6.1 Final Statement 8

3 Lab 2: Mixin Implementation 8

3.1 Exercise: Mixin Implementation 8
3.1.1 Task 1: Implementation 8
3.1.2 Questions / Exercises 9
3.1.3 Task 2: Attribute Enhancement 9
3.1.4 Questions / Exercises 10
3.1.5 Task 3: Initialization 10
3.1.6 Questions / Exercises 11
3.1.7 Task 4: Methods 11
3.1.8 Questions / Exercises 11
3.2 Final Questions 11

4 Lab Summary 11

1 Goal

Mixins are powerful, but they are no universal solution to every design problem. Purpose of this HOL is to teach **when** to use mixins and **why**.

The reference sample to describe a use case for mixins is the implementation of equality with .NET. We will try to solve this problem with several approaches. We will show in which situations other approaches might not be appropriate and highlight some of their limitations.

The final implementation for this use case is done with mixins to depict which problems are solved with mixin for this and similar use cases.

This HOL contains sections with questions. Some answers might be straight forward. Other questions could be tricky. We encourage you to talk about your ideas and your proposed solutions with other developers.

Please have also a look on the further readings section in the end of this HOL. They provide some suggestions on how you can proceed to learn more.

Recommendation: There is a lot of source code available in this HOL that can be copied and pasted. It might be helpful to typewrite some parts to get a more focused view on the code.

Topics:

- Basic implementation of equality with .NET
- Implementing equality with inherited types
- Implementing static utility methods to solve the equality
- Implementing a strategy pattern to offer several “equals strategies”
- Developing a “EquatableByValues” Mixin

Important:

For t his HOL Visual Studio 2010 must be installed on your working PC.

We recommend installing the following two third party tools too:

- JetBrains Resharper (http://www.jetbrains.com/resharper/)
- RedGate Reflector (http://reflector.red-gate.com/Download.aspx)

Both tools will help you to understand the samples in this HOL better.

Go to https://www.re-motion.org/builds/ to get the latest re-motion build. All samples were tested with version 1.13.95.

2 Lab 1: Equals without Mixins

Estimated Time: 30 minutes

2.1 Theory: Equals

It is essential to understand the background of “equality” to solve this HOL. The following two articles provide a good instruction.

- http://msdn.microsoft.com/en-us/library/bsc2ak47.aspx
- http://www.codeproject.com/KB/dotnet/IEquatable.aspx

Get started once you are sure that you know the

- difference between reference types and value types
- difference between reference equality and bitwise equality

2.2 Exercise 1: Basic Equals Implementation

Please note: To keep implementation small, we implement each approach side by side in one console application and use namespaces to separate them. This is not a recommended practice in projects, but it is perfectly acceptable for a show case.

2.2.1 Task 1: Implementation

1. **Start Visual Studio 2010**
2. **Add a new project and select console application. Use the following settings:**

   Name: EqualsShowCase

   Location: C:\HOL\mixins

   Solution name: HOLEquals
3. **Add a file BasicImplementation.cs to the project and use the following source code**

```
namespace EqualsShowCase.BasicImplementation
{
  public class Address
  {
    public string Street;
    public string StreetNumber;
    public string City;
    public string ZipCode;
    public string Country;

    public override bool Equals (object obj)
    {
      return Equals (obj as Address);
    }

    public bool Equals (Address other)
    {
      if (other == null)
        return false;

      if (this.GetType () != other.GetType ())
        return false;

      return Street == other.Street && StreetNumber == other.StreetNumber &&
             City == other.City && ZipCode == other.ZipCode && Country == other.Country;
    }

    public override int GetHashCode ()
    {
      return Street.GetHashCode () ^ StreetNumber.GetHashCode () ^
             City.GetHashCode () ^ ZipCode.GetHashCode () ^ Country.GetHashCode ();
    }
  }
}
```

4. **Replace the existing code of program.cs with the following source code.**

```
using System;

namespace HOLApp
{
  class Program
  {
    static void Main (string[] args)
    {
      BasicImplementation();
      // InheritanceImplementation();
      // UtilityImplementation();
      // StrategyImplementation();
      // MixinImplementation();

      Console.ReadKey();
    }
    static void BasicImplementation ()
    {
      EqualsShowCase.BasicImplementation.Address address1 = new
EqualsShowCase.BasicImplementation.Address ();
      EqualsShowCase.BasicImplementation.Address address2 = new
EqualsShowCase.BasicImplementation.Address ();
      Console.WriteLine ("Basic Implementation: Both instances have the same values: {0}", Equals
(address1, address2));
    }
  }
}
```

5. **Debug and verify the results**
6. **Change the class declaration to**

```
public class Address : IEquatable<Address>
```

7. **Debug and verify the results**

2.2.2 Questions / Excercises

- It would be possible to implement Equals without overriding GetHashCode(). In which problems could you run into?
- There are five fields in class Address. There could be far more field. Is there a generic way to solve get the values of all fields?
- Does it make sense to define the equality of two types by comparing their property values too?
- Can you think of other “common value equals implementations” besides comparing the fields of two object instances? (Two objance instances are equal in case … are equal)
- We have added the interface IEquatable<Address> in step 6. Please explain why this makes sense?
- If we implement a class PhoneNumber with fields such as CountryCode, AreaCode, Number, Extension, we have to implement a similar “equate by field values” method too. True or false: The *second* implementation of Equals () would be then a perfect example for boilerplate code?
- Preparation for Exercise 2: If you want to apply the similar functionality to other classes, what can you do to avoid implementing the functionality to compare the values of the fields?
- **Expert Question:** Using DDD terminology, class Address is a perfect example for a *Value Object* (do not confuse this terminology not with a value type). Should there be a different approach to determine equality with *Entities* and *Aggregates*?

2.3 Theory: Reflection

If you have no experience with reflection, we recommend that you get an overview on this topic. A good starting point might be:

http://msdn.microsoft.com/en-us/library/f7ykdhsy(v=VS.100).aspx

2.4 Exercise 2: Using Inheritance

In this exercise we try to add inheritance to our sample as we do not want to implement the same functionality again for other types that would use the same “equals strategy”.

2.4.1 Task 1: Implementation

1. **Add a new file InheritanceImplementation.cs and use the following code**

```
using System;
using System.Linq;
using System.Reflection;

namespace EqualsShowCase.InheritanceImplementation

{
  public class EquatableByValues<T> : IEquatable<T>
```

```
     where T : class
  {
     private static readonly FieldInfo[] s_targetFields = typeof (T).GetFields (
                  BindingFlags.Instance | BindingFlags.Public | BindingFlags.NonPublic);

    public bool Equals (T other)
    {
      if (other == null)
        return false;

      if (GetType () != other.GetType ())
        return false;

      for (int i = 0; i < s_targetFields.Length; i++)
      {
        object thisFieldValue = s_targetFields[i].GetValue (this);
        object otherFieldValue = s_targetFields[i].GetValue (other);

        if (!Equals (thisFieldValue, otherFieldValue))
          return false;
      }

      return true;
    }

    public new int GetHashCode ()
    {
      return s_targetFields.Aggregate (0, (current, t) => current ^ t.GetValue (this).GetHashCode
());
    }
  }
}
```

2. **Add the following class to this namespace**

```
  public class Address : EquatableByValues<Address>
  {
    public string Street;
    public string StreetNumber;
    public string City;
    public string ZipCode;
    public string Country;
  }
```

3. **Add the following code to the program.cs**

```
    static void InheritanceImplementation ()
    {
      EqualsShowCase.InheritanceImplementation.Address address3 = new
EqualsShowCase.InheritanceImplementation.Address ();
      EqualsShowCase.InheritanceImplementation.Address address4 = new
EqualsShowCase.InheritanceImplementation.Address ();
      Console.WriteLine ("Inheritance Implementation: Both instances have the same values: {0}",
Equals (address3, address4));
}
```

4. **Uncomment `// InheritanceImplementation();` in the Main method and test**
5. **Change the implementation of InheritanceImplementation.Address as follows**

```
public class Address : EquatableByValues<Address>
  {
    public string City;
    public string ZipCode;
    public string Country;
  }

  public class StreetAddress : Address
  {
    public string Street;
    public string StreetNumber;
  }
```

6. **Change the InheritanceImplementation logic in program.cs**

```
    static void InheritanceImplementation ()
    {
      EqualsShowCase.InheritanceImplementation.Address address3 = new
EqualsShowCase.InheritanceImplementation.Address ();
      EqualsShowCase.InheritanceImplementation.Address address4 = new
EqualsShowCase.InheritanceImplementation.Address ();
      Console.WriteLine ("Inheritance Implementation: Both instances have the same values: {0}",
Equals (address3, address4));

      EqualsShowCase.InheritanceImplementation.StreetAddress streetaddress1 = new
EqualsShowCase.InheritanceImplementation.StreetAddress ();
```

```
      EqualsShowCase.InheritanceImplementation.StreetAddress streetaddress2 = new
EqualsShowCase.InheritanceImplementation.StreetAddress ();

      streetaddress1.Street = "Test";
      streetaddress2.Street = "Test2";

      // Value is not as might be expected
      Console.WriteLine ("(Value not as expected)Inheritance Implementation StreetAddress: Both
instances have the same values: {0}", Equals (streetaddress1, streetaddress2));
    }
```

7. **Debug and Test**

2.4.2 Questions / Excercises

- Problem 1: Look at **`public class Address : EquatableByValues<Address>`**. In which ways are you restricted? What if Address shall also derive from other classes? What about “Multiple Inheritance in .NET”?
- Problem 2: Why does the “StreetAddress Equals()” does not return the expected values? Hint: Which interface does StreetAddress implement?
- We are using reflection in this sample. Do you think this can lead to performance issues?

2.5 Exercise: Utility Implementation

2.5.1 Task 1: Implementation

1. **Add a new file UtilityImplementation.cs**
2. **Use the following code for the file**

```
using System;
using System.Reflection;

namespace EqualsShowCase.UtilityImplementation
{
  public class EquateUtility
  {
    public static bool EquateByValues (object a, object b)
    {
      if ((a == null) && (b == null))
        return true;

      if ((a == null) || (b == null))
        return false;

      if (a.GetType () != b.GetType ())
        return false;

      FieldInfo[] s_targetFields = a.GetType().GetFields (BindingFlags.Instance |
BindingFlags.Public | BindingFlags.NonPublic);

      for (int i = 0; i < s_targetFields.Length; i++)
      {
        object thisFieldValue = s_targetFields[i].GetValue (a);
        object otherFieldValue = s_targetFields[i].GetValue (b);

        if (!Equals (thisFieldValue, otherFieldValue))
          return false;
      }

      return true;
    }
    public static int GetHashCode (object a)
    {
      int i = 0;
      foreach (FieldInfo f in a.GetType().GetFields (BindingFlags.Instance | BindingFlags.Public
| BindingFlags.NonPublic))
        i = i ^ f.GetHashCode ();
      return i;
    }
  }

  public class Address : IEquatable<Address>
  {
    public string City;
    public string ZipCode;
    public string Country;
    public override bool Equals (object obj)
    {
      return Equals (obj as Address);
    }

    public bool Equals (Address other)
    {
      return EqualsShowCase.UtilityImplementation.EquateUtility.EquateByValues (this, other);
    }

    public override int GetHashCode ()
    {
      return EqualsShowCase.UtilityImplementation.EquateUtility.GetHashCode (this);
    }
  }

  public class StreetAddress : Address
  {
    public string Street;
    public string StreetNumber;
  }
}
```

3. **Add a new a new static method to your program.cs and call it.**

```
    static void UtilityImplementation ()
    {
      EqualsShowCase.UtilityImplementation.StreetAddress streetaddress3 = new
EqualsShowCase.UtilityImplementation.StreetAddress ();
      EqualsShowCase.UtilityImplementation.StreetAddress streetaddress4 = new
EqualsShowCase.UtilityImplementation.StreetAddress ();
      streetaddress3.Street = "Test";
      streetaddress4.Street = "Test2";
      Console.WriteLine ("Utility Implementation StreetAddress: Both instances have the same
values: {0}", Equals (streetaddress3, streetaddress4));
    }
```

4. **Uncomment `// UtilityImplementation ()` in the Main method**

2.5.2 Questions / Exercises

- Statement: “In many projects there is a “UtilityHell”. Step by step this assembly gets stuffed and nobody knows if some implementations are still used and by whom”.
    - Bad case: Every developer puts generic functionality in static methods of a common Utility Assembly.
    - Worse Case: A developer does not even check if there is already an existing Utility implementation available.
    - Worst case: A developer checks if a similar implementation available and if he finds a similar implementation, he changes the existing implementation to his own needs (without looking who was calling this method too).

  What is your opinion about this statement? Have you encountered something similar?
- Step by step this assembly gets stuffed and nobody knows if some implementations are still used and by whom”. What is your opinion about this statement? Have you encountered something similar?
- Discuss the following enhancement

```
    protected delegate bool EqualCheck (object a, object b);
    protected virtual EqualStrategy GetEqualsStrategy ()
    {
      return EqualsShowCase.UtilityImplementation.EqualsUtility.FieldEquals;
    }

    public bool Equals (Address other)
    {
      return GetComparison () (this, other);
    }
```

- It might be necessary to store state information. We might want to store that two objects would be non-equal, if they were case insensitive. How can this be implemented with this approach?

2.6 Exercise: Strategy Implementation

1. **Add a file StrategyPatternImplementation.cs und use the following code**

```
using System;
using System.Reflection;

namespace EqualsShowCase.StrategyImplementation
{
  public interface IEqualsStrategy
  {
    bool CustomEquals(object a, object b);
  }

  public class EquatableByValueStrategy : IEqualsStrategy
  {
    public bool CustomEquals (object a, object b)
    {
      if ((a == null) && (b == null))
        return true;

      if ((a == null) || (b == null))
        return false;

      if (a.GetType () != b.GetType ())
        return false;

      FieldInfo[] s_targetFields = a.GetType ().GetFields (BindingFlags.Instance |
BindingFlags.Public | BindingFlags.NonPublic);

      for (int i = 0; i < s_targetFields.Length; i++)
      {
        object thisFieldValue = s_targetFields[i].GetValue (a);
        object otherFieldValue = s_targetFields[i].GetValue (b);

        if (!Equals (thisFieldValue, otherFieldValue))
          return false;
      }

      return true;
    }
  }

  public class Address : IEquatable<Address>
  {
    private IEqualsStrategy _equalsStrategy;
    public string City;
    public string ZipCode;
    public string Country;

    public Address ()
    {
      _equalsStrategy = new EquatableByValueStrategy();
    }
    public Address (IEqualsStrategy equalsStrategy)
    {
      _equalsStrategy = equalsStrategy;
    }

    public override bool Equals (object obj)
    {
      return Equals (obj as Address);
    }

    public bool Equals (Address other)
    {
      return _equalsStrategy.CustomEquals(this, other);
    }

    public override int GetHashCode ()
    {
      return City.GetHashCode () ^ ZipCode.GetHashCode () ^ Country.GetHashCode ();
    }
  }

  public class StreetAddress : Address
  {
    public string Street;
    public string StreetNumber;
  }
}
```

2. **Add the following code to program.cs und uncomment // StrategyImplemention()**

```
    private static void StrategyImplementation ()
    {
      EqualsShowCase.StrategyImplementation.StreetAddress streetaddress3 = new
EqualsShowCase.StrategyImplementation.StreetAddress ();
      EqualsShowCase.StrategyImplementation.StreetAddress streetaddress4 = new
EqualsShowCase.StrategyImplementation.StreetAddress ();
      streetaddress3.Street = "Test";
      streetaddress4.Street = "Test2";
      Console.WriteLine ("Strategy Implementation StreetAddress: Both instances have the same
values: {0}", Equals (streetaddress3, streetaddress4));
    }
```

2.6.1 Final Statement

- With help of the strategy pattern, a developer is able to define the way an object instance is equated.

  But in this implementation a developer still has to override Equals() and GetHashCode() to call the corresponding interface methods. There is a lot of boiler plate code.

  In addition to this, the implementation is bound to the interface. Let's say you want to be able to pass parameters to enable/disable case sensitive comparison for strings or to detect abbreviations (Wall Street should be equal to same as Wall Str.) for your custom implementation. For this functionality you probably would need a new interface that supports something like: **`bool Equals(object first, object second, bool checkCaseSensitive, bool detectAbbrevations).`**

3 Lab 2: Mixin Implementation

Estimated Time: 15 minutes

3.1 Exercise: Mixin Implementation

3.1.1 Task 1: Implementation

1. **Add the following references to the projects**

- remotion.dll
- remotion.interfaces.dll

Please note: A good practice might be to put all re-motion assemblies under the directory References below the solution file.

1. **Add a file MixinImplementation.cs and use the following code**

```
using System;
using System.Linq;
using System.Reflection;
using Remotion.Mixins;

namespace EqualsShowCase.MixinImplementation
{
  public class EquatableByValuesMixin<T> : Mixin<T>, IEquatable<T>
     where T : class
  {
    private static readonly FieldInfo[] s_targetFields = typeof (T).GetFields (
              BindingFlags.Instance | BindingFlags.Public | BindingFlags.NonPublic);

    bool IEquatable<T>.Equals (T other)
    {
      if (other == null)
        return false;

      if (Target.GetType () != other.GetType ())
        return false;

      for (int i = 0; i < s_targetFields.Length; i++)
      {
        object thisFieldValue = s_targetFields[i].GetValue (Target);
        object otherFieldValue = s_targetFields[i].GetValue (other);

        if (!Equals (thisFieldValue, otherFieldValue))
          return false;
      }
      return true;
    }

    [OverrideTarget]
    public new bool Equals (object other)
    {
      return ((IEquatable<T>)this).Equals (other as T);
    }

    [OverrideTarget]
    public new int GetHashCode ()
    {
      int i = 0;
      foreach (FieldInfo f in s_targetFields)
        i = i ^ f.GetHashCode();
      return i;
    }
  }

  public class Address
  {
    public string City;
    public string ZipCode;
    public string Country;
  }

  public class StreetAddress : Address
  {
    public string Street;
    public string StreetNumber;
  }
}
```

2. **Add the following source code to program.cs and uncomment // MixinImplementation()**

```
    static void MixinImplementation ()
    {
      EqualsShowCase.MixinImplementation.StreetAddress mixedAddress1 =
ObjectFactory.Create<EqualsShowCase.MixinImplementation.StreetAddress> (ParamList.Empty);
      EqualsShowCase.MixinImplementation.StreetAddress mixedAddress2 =
ObjectFactory.Create<EqualsShowCase.MixinImplementation.StreetAddress> (ParamList.Empty);
      Console.WriteLine ("Mixed Implementation StreetAddress: Both instances have the same
values: {0}", Equals (mixedAddress1, mixedAddress2));
    }
```

3. **Enable Mixins by "class decoraction"**

```
[Uses (typeof (EquatableByValuesMixin<Address>))]
public class Address
```

4. **Build and test**

3.1.2 Questions / Exercises

- Have a look on the source code and identify the parts that reference re-motion assemblies. Look them up in the source code or inspect them via intellisense (pdb files required).
- Mixin<T> is required to reference the Target. If no "Target is needed", is it possible to remove this base class?

3.1.3 Task 2: Attribute Enhancement

1. **Replace `[Uses (typeof (EquatableByValuesMixin<Address>))]` with**

```
 [Uses (typeof (EquatableByValuesMixin<>))]
```

2. **Build and debug**
3. **Replace with `public class EquatableByValuesMixin<T> : Mixin<T>, IEquatable<T>` with**

```
public class EquatableByValuesMixin<[BindToTargetType]T> : Mixin<T>, IEquatable<T>
```

4. **Build and debug**
5. **Add**

```
  public class EquatableByValuesAttribute : UsesAttribute
  {
    public EquatableByValuesAttribute ()
      : base (typeof (EquatableByValuesMixin<>))
    {
    }
  }
```

6. **Replace `[Uses (typeof (EquatableByValuesMixin<>))]` with**

```
[EquatableByValuesAttribute]
```

7. **`Build and debug`**

3.1.4 Questions / Exercises

- Use Reflector or the re-motion source code to investigate **`BindToTargetType`**

3.1.5 Task 3: Initialization

1. **Replace class Address of the MixinImplementation namespace with the following code**

```
public abstract class Address
  {
    protected Address ()
    {
    }

    protected Address (int zipCode, string city)
    {
      ZipCode = zipCode;
      City = city;
    }

    protected Address (int zipCode, City city)
    {
      ZipCode = zipCode;
      City = city.Name; // null-check missing
    }

    public int ZipCode;
    public string City;
  }

  public class City
  {
    public string Name;
  }

  public class StreetAddress : Address
  {
    public StreetAddress (int zipCode, string city, string street, string streetNumber)
      : base (zipCode, city)
    {
      Street = street;
      StreetNumber = streetNumber;
    }

    public string Street;
    public string StreetNumber;
  }

  public class POBoxAddress : Address
  {
    public POBoxAddress (int zipCode, string city, int poBox)
      : base (zipCode, city)
    {
      POBox = poBox;
    }

    public int POBox;
  }
```

2. **Replace the current implementation with**

```
static void MixinImplementation ()
    {
      EqualsShowCase.MixinImplementation.StreetAddress mixedAddress1 =
ObjectFactory.Create<EqualsShowCase.MixinImplementation.StreetAddress> (ParamList.Create (1010,
"Wien", "Stephansplatz", "1"));
      EqualsShowCase.MixinImplementation.StreetAddress mixedAddress2 =
ObjectFactory.Create<EqualsShowCase.MixinImplementation.StreetAddress>(ParamList.Create(1010,
"Wien", "Stephansplatz", "1"));
      Console.WriteLine ("Mixed Implementation StreetAddress: Both instances have the same
values: {0}", Equals (mixedAddress1, mixedAddress2));
```

```
      EqualsShowCase.MixinImplementation.StreetAddress mixedAddress3 =
ObjectFactory.Create<EqualsShowCase.MixinImplementation.StreetAddress> (ParamList.Create (1010,
default (string), "Stephansplatz", "1"));
      EqualsShowCase.MixinImplementation.StreetAddress mixedAddress4 =
ObjectFactory.Create<EqualsShowCase.MixinImplementation.StreetAddress> (ParamList.Create (1010,
new City().Name = "Wien", "Stephansplatz", "1"));
      Console.WriteLine ("Mixed Implementation StreetAddress: Both instances have the same
values: {0}", Equals (mixedAddress1, mixedAddress3));
      Console.WriteLine ("Mixed Implementation StreetAddress: Both instances have the same
values: {0}", Equals (mixedAddress1, mixedAddress4));
    }
```

3.1.6 Questions / Exercises

- ParamList.Create has a huge list of overloads. Can you explain why this might be necessary?

3.1.7 Task 4: Methods

1. Change `bool IEquatable<T>.Equals (T other)` to

```
public bool Equals (T other)
```

2. Change `return ((IEquatable<T>) this).Equals (other as T);` to

```
return Equals (other as T);
```

3.1.8 Questions / Exercises

- Can you explain what has changed in Task 3?
- Does it make sense to make **`public book Equals(T other)`** virtual?

3.2 Final Questions

- In which other uses cases it might make sense to use mixins?
- Consider the following scenario:

  You have several Entities as Domain Objects. Two of them are: Person and Document. In your business application, you need the following generic functionality:

  - Revision Tracking,
  - Change Tracking
  - Approval.

  The traditional way to implement this is to add a “BaseDomainObject class” that contains generic code for these three use cases.

  But not all Domain Object need “everything”. There is no approval for changes to a person required, whereas approval is obligatory to Documents. Does it make sense to replace inheritance with mixins, such as (remark: **`ComposedObject`** is a pattern to include more interfaces in one base class).

```
  [Uses(typeof(TrackableMixin))]
  [Uses(typeof(RevisionableMixin))]
  public class Person : ComposedObject<IPerson>, IPersonImpl

  [Uses(typeof(TrackableMixin))]
  [Uses(typeof(ApprovableMixin))]
  [Uses(typeof(RevisionableMixin))]
  public class Document : ComposedObject<IDocument>, IDocumentImpl
```

4 Lab Summary

You have successfully done the following:

- You have implemented several “known approaches” to solve equality in .NET
- You have implemented a mixin with a “uses scenario” to solve equality in .NET

You are now aware **when** and **why** you can use a mixin in a common use case. You might want to:

- Read the blogs on www.re-motion.org to learn more about implementation details and additional features how mixins can be implemented.
- Investigate the “Extends Scenario” of mixins (You can get basic information in the other HOLs for Mixins).
- Read “Head First Design Patterns” to be able to understand design techniques in total better. It helps you to put mixins in relation to other design patterns.
- Research on Composite Oriented Programming(COP) and DCI and see how mixins are related to these development strategies.
- Read the next re-motion HOL to understand another re-motion technology.
- On http://en.wikipedia.org/wiki/Mixin in further readings, there are many references to other mixin implementations, you might want to read them and compare them with the re-motion approach.
- Try to implement mixins with Ruby
- For those who want to become real experts: Read “CLR via C#”. This book gives you a deep dive into the .NET Framework and you will be able to understand how mixins are implemented in re-mix. Attention: It may take also developers with a good background experience a lot of time to fully understand the content of that book. Reserve an extended research time and prepare enough coffee.