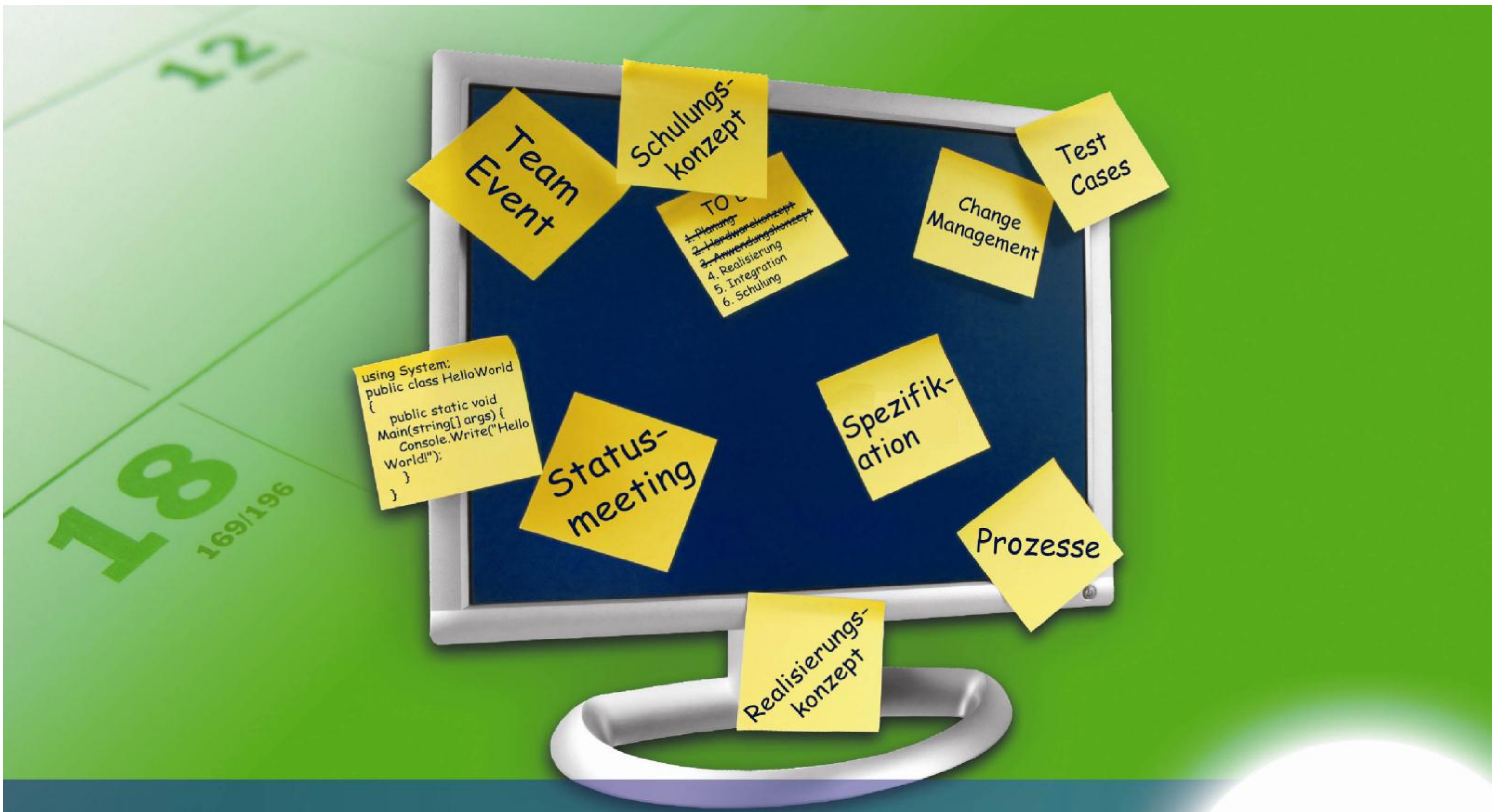

Mixin Show Case: Equals

Hands on Labs

rubicon
informationstechnologie gmbh

Table of Contents

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 use case” in this HOL is the implementation of equality with .NET. We will try to solve this problem with several approaches. We will show why other approaches might not be the best solution and how they are limited.

The final implementation 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 direct into Visual Studio to get a more focused view on the code.

Topics:

- Implement equality with .NET
- Add inheritance to the first implementation
- Offer a common equals strategy for objects with static utility methods
- Refine the utility method approach by using the strategy design pattern
- Develop a Mixin to solve the problem

Important:

For this HOL Visual Studio 2010 must be installed on your working PC. For all examples the re-mix binaries are required. You can download compiled binaries and source code from codeplex (http://remix.codeplex.com)

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 introduction.

- 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
- reference equality and bitwise equality

2.2 Exercise 1: Basic Equals Implementation

Please note: To keep our 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 EqualsShowCase
{
  class Program
  {
    static void Main (string[] args)
    {
      BasicImplementation ();
      //InheritanceImplementation ();
      //UtilityImplementation ();
      //StrategyImplementation ();
      //MixinImplementation ();

      Console.ReadKey ();
    }

    static void BasicImplementation ()
    {
      var wien1 = new EqualsShowCase.BasicImplementation.Address { City = "Wien" };
      var wien2 = new EqualsShowCase.BasicImplementation.Address { City = "Wien" };
      var berlin = new EqualsShowCase.BasicImplementation.Address { City = "Berlin" };

      Console.WriteLine ("Basic Implementation: Wien = Wien: {0}", Equals (wien1, wien2));
      Console.WriteLine ("Basic Implementation: Wien = Berlin: {0}", Equals (wien1, berlin));
    }
  }
}
```

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 / Exercises

- It is possible to implement **`Equals()`** without overriding **`GetHashCode()`.** In which problems could you run into?
- There are five fields in class Address. Is there a generic way to compare each field per class?
- Does it make sense to compare the values of properties too to determine if two objects are equal?
- Can you think of other “value equals implementations” besides comparing the fields of two object instances? (Two object instances are equal in case … are equal)
- We added an implementation of the interface IEquatable<Address> in Step 6. Please explain why this makes sense?
- If we create a class PhoneNumber with fields such as CountryCode, AreaCode, Number, Extension, we might have to implement a similar “equate by field values” method too. True or false: Implementing a similar implementation a second time is an example for creating boilerplate code?
- Preparation for Exercise 2: If you want to apply the similar “equals functionality” to other classes, what can you do to avoid implementing the functionality to compare the values of the fields a second time?
- **Expert Question:** Class Address is a perfect example for a *value object* in the DDD terminology(do not confuse value objects with a value types). 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 add inheritance to our sample to avoid implementing the same functionality again for other types that would require 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.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 override bool Equals (object obj)
  {
    return Equals (obj as T);
  }

  public override int GetHashCode ()
  {
    int i = 0;
    foreach (FieldInfo f in s_targetFields)
      i ^= f.GetValue (this).GetHashCode ();
    return i;
  }
}
```

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 ()
{
  var wien1 = new EqualsShowCase.InheritanceImplementation.Address { City = "Wien" };
  var wien2 = new EqualsShowCase.InheritanceImplementation.Address { City = "Wien" };

  var berlin = new EqualsShowCase.InheritanceImplementation.Address { City = "Berlin" };
  Console.WriteLine ("Inheritance Implementation: Wien = Wien: {0}", Equals (wien1, wien2));
  Console.WriteLine ("Inheritance Implementation: Wien = Berlin: {0}", Equals (wien1, berlin));
}
```

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 ()
{
  var wien1 = new EqualsShowCase.InheritanceImplementation.Address { City = "Wien" };
  var wien2 = new EqualsShowCase.InheritanceImplementation.Address { City = "Wien" };

  var berlin = new EqualsShowCase.InheritanceImplementation.Address { City = "Berlin" };
  Console.WriteLine ("Inheritance Implementation: Wien = Wien: {0}", Equals (wien1, wien2));
  Console.WriteLine ("Inheritance Implementation: Wien = Berlin: {0}", Equals (wien1, berlin));

  var kaerntnerStrasse1 = new EqualsShowCase.InheritanceImplementation.StreetAddress { City =
"Wien", Street = "Kärntner Straße" };
  var kaerntnerStrasse2 = new EqualsShowCase.InheritanceImplementation.StreetAddress { City =
"Wien", Street = "Kärntner Straße" };
```

```
  var amGraben = new EqualsShowCase.InheritanceImplementation.StreetAddress { City = "Wien",
Street = "Am Graben" };
  Console.WriteLine ("Inheritance Implementation: Kärntner Straße = Kärntner Straße: {0}", Equals
(kaerntnerStrasse1, kaerntnerStrasse2));
  Console.WriteLine ("Inheritance Implementation: Kärntner Straße = Am Graben: {0} (should be
false!)", Equals (kaerntnerStrasse1, amGraben));
}
```

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 “StreetAddress.Equals()” return false? 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

The strategy to equate two objects can also be independent from class inheritance strategies. In the following sample, we want to move the logic to an independent utility method.

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 EqualsByValues (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[] targetFields = a.GetType ().GetFields (BindingFlags.Instance |
BindingFlags.Public | BindingFlags.NonPublic);
      for (int i = 0; i < targetFields.Length; i++)
      {
        object thisFieldValue = targetFields[i].GetValue (a);
        object otherFieldValue = targetFields[i].GetValue (b);

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

      return true;
    }

    public static int GetHashCodeByValues (object obj)
    {
      FieldInfo[] targetFields = obj.GetType ().GetFields (BindingFlags.Instance |
BindingFlags.Public | BindingFlags.NonPublic);
      int j = 0;
      foreach (var f in targetFields)
      {
        j ^= f.GetValue (obj).GetHashCode ();
      }
      return j;
    }
  }

  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 EquateUtility.EqualsByValues (this, other);
    }

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

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

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

```
static void UtilityImplementation ()
{
  var wien1 = new EqualsShowCase.UtilityImplementation.StreetAddress { City = "Wien" };
  var wien2 = new EqualsShowCase.UtilityImplementation.StreetAddress { City = "Wien" };
  var berlin = new EqualsShowCase.UtilityImplementation.StreetAddress { City = "Berlin" };
  Console.WriteLine ("Utility Class Implementation: Wien = Wien: {0}", Equals (wien1, wien2));
  Console.WriteLine ("Utility Class Implementation: Wien = Berlin: {0}", Equals (wien1, berlin));
}
```

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

2.5.2 Questions / Exercises

- Statement: “In many projects there is a “UtilityHell”. Step by step utility classes get stuffed. After some time, nobody knows if some implementations are still used and by whom”.
  - Bad case: Every developer puts generic functionality in static methods of Utility classes. There are classes such as CompareUtility, FileUtility, etc.
  - Worse Case: A developer does not 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 is calling this method).

  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.EqualsByValues;
    }

    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 in “utility methods”?

2.6 Exercise: Strategy Implementation

The utility methods are very often used by people who are not aware how design patterns can solve some typical OOP issues. A better way to approach a situation where people move code into static methods is the strategy pattern.

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 first, object second);
     int CustomGetHashCode (object obj);
   }

   public class EquatableByValueStrategy<T> : IEqualsStrategy
   {
    static FieldInfo[] s_targetFields = typeof(T).GetFields (BindingFlags.Instance |
BindingFlags.Public | BindingFlags.NonPublic);

    public bool CustomEquals (object first, object second)
    {
     if ((first == null) && (second == null))
       return true;

     if ((first == null) || (second == null))
       return false;

     if (first.GetType () != second.GetType ())
       return false;

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

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

     return true;
    }

   public int CustomGetHashCode (object obj)
   {
     int i = 0;
     foreach (FieldInfo f in s_targetFields)
       i ^= f.GetValue (obj).GetHashCode ();
     return i;
   }
  }

  public class Address : IEquatable<Address>
  {
    private IEqualsStrategy _equalsStrategy;

    public string City;
    public string ZipCode;
    public string Country;

    public Address () : this (new EquatableByValueStrategy<Address>())
    {}

    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 _equalsStrategy.CustomGetHashCode (this);
    }
 }

  public class StreetAddress : Address
  {
    public StreetAddress () : this (new EquatableByValueStrategy<StreetAddress>())
    {}
    public StreetAddress (IEqualsStrategy equalsStrategy ) : base (equalsStrategy)
    {}

    public string Street;
    public string StreetNumber;
  }

}
```

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

```
    private static void StrategyImplementation ()
    {
      var wien1 = new EqualsShowCase.StrategyImplementation.StreetAddress { City = "Wien" };
      var wien2 = new EqualsShowCase.StrategyImplementation.StreetAddress { City = "Wien" };
      var berlin = new EqualsShowCase.StrategyImplementation.StreetAddress { City = "Berlin" };

      Console.WriteLine ("Strategy Implementation: Wien = Wien: {0}", Equals (wien1, wien2));
      Console.WriteLine ("Strategy Implementation: Wien = Berlin: {0}", Equals (wien1, berlin));
    }
```

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 "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: The assemblies can be downloaded from remix.codeplex.com. A good practice might be to put all 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 ^= f.GetValue (Target).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**

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

3. **uncomment // MixinImplementation()**
4. **Enable Mixins by "class decoraction"**

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

5. **Build and test**

3.1.2 Questions / Exercises

- Identify the keywords of MixinImplemention.cs that are part of re-mix assemblies. Look them up in the re-mix source code or inspect the DLLs via Intellisense (pdb files required). What is your conclusion?
- Mixin<T> is required to reference the Target class. If no "Target is needed", is it possible to remove this base class?
- The re-mix assemblies have references to other assemblies (such as Microsoft.Practices.ServiceLocation, Castle.Core, Castle.DynamicProxy2). Find out what these assemblies do!

3.1.3 Task 2: Attribute Enhancement

In the following task we will optimize the attribute usage for the Mixin implementation. We will show that there are more elegant ways to decorate a class as a mixin.

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!`** What does this attribute do?

3.1.5 Task 3: Initialization

In this task we will change the implementation of Address to improve the object initialization.

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

```
[EquatableByValues]
public abstract class Address
{
  protected Address ()
  {
  }

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

  // This alternative constructor shows the problem of runtime overloads. Assuming you create an
Address using the statements
  //   string city;
  //   ObjectFactory.Create (1010, city)
  // and the Create method would accept a params object[], the implementation would pick the
correct constructor overload only
  // if city != null!
  // ParamList.Create is type-safe in this regard and will always pick the constructor overload
using the static types of
  // the arguments (in this case, int and string)
  protected Address (int zipCode, City city)
  {
    ZipCode = zipCode;
    City = city != null ? city.Name : null;
  }

  public int ZipCode;
  public string City;
}

public class City
{
  public string Name;
}

public class StreetAddress : Address
{
  public static StreetAddress NewObject (int zipCode, string city, string street, string
streetNumber)
  {
    return ObjectFactory.Create<StreetAddress> (true, ParamList.Create (zipCode, city, street,
streetNumber));
  }

  protected 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 static POBoxAddress NewObject (int zipCode, string city, int poBox)
  {
    return ObjectFactory.Create<POBoxAddress> (true, ParamList.Create (zipCode, city, poBox));
  }

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

  public int POBox;
}
```

2. **Replace the current implementation with**

```
    static void MixinImplementation ()
    {
      var wien1 = EqualsShowCase.MixinImplementation.StreetAddress.NewObject (1010, "Wien",
"Stephansplatz", "1");
      var wien2 = EqualsShowCase.MixinImplementation.StreetAddress.NewObject (1010, "Wien",
"Stephansplatz", "1");
      var berlin = EqualsShowCase.MixinImplementation.StreetAddress.NewObject (11011, "Berlin",
"Pariser Platz", "1");

      Console.WriteLine ("Mixin Implementation: Wien = Wien: {0}", Equals (wien1, wien2));
      Console.WriteLine ("Mixin Implementation: Wien = Berlin: {0}", Equals (wien1, berlin));

      // show that the correct constructor overload is called even if the type of the city
parameter cannot
      // be detected at runtime (could be string or City). Would result in an AmbiguityException
otherwise
      var nullCitymixedAddress3 = EqualsShowCase.MixinImplementation.StreetAddress.NewObject
(1010, null, "Stephansplatz", "1");
      Console.WriteLine ("Mixin Implementation: Wien = null: {0}", Equals (wien1, null));
    }
```

3.1.6 Questions / Exercises

- ParamList.Create has a huge list of overloads. Can you explain why they are required?

3.1.7 Task 4: Equals-Methods

In the final task we will show adapt how to call the Equals 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 4?
- Does it make sense to make **`public book Equals(T other)`** virtual?

3.2 Final Questions

- In which other uses cases might it make sense to use mixins?
- We have introduced a uses scenario with this sample. In simple words: In a uses scenario the Target class is decorated with a Uses attribute to specify that the Target class uses the specified Mixin(s). In the extends scenario, the Mixin is decorated with an Extends attribute to specify that the Mixin extends the specified Target class(es). In other words: In the uses scenario the class knows

its mixins, in the extends scenario the mixin knows its classes. Can you think of a good use case for this extends scenario?

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 well aware of a situation where mixins can be a perfect solution for a common problem and you know how to solve it with mixins.

If you want to do more, here are some recommendations:

- 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.
- 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 Data Context Interaction (DCI) and see how mixins are related to these development strategies.
- 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 even 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.