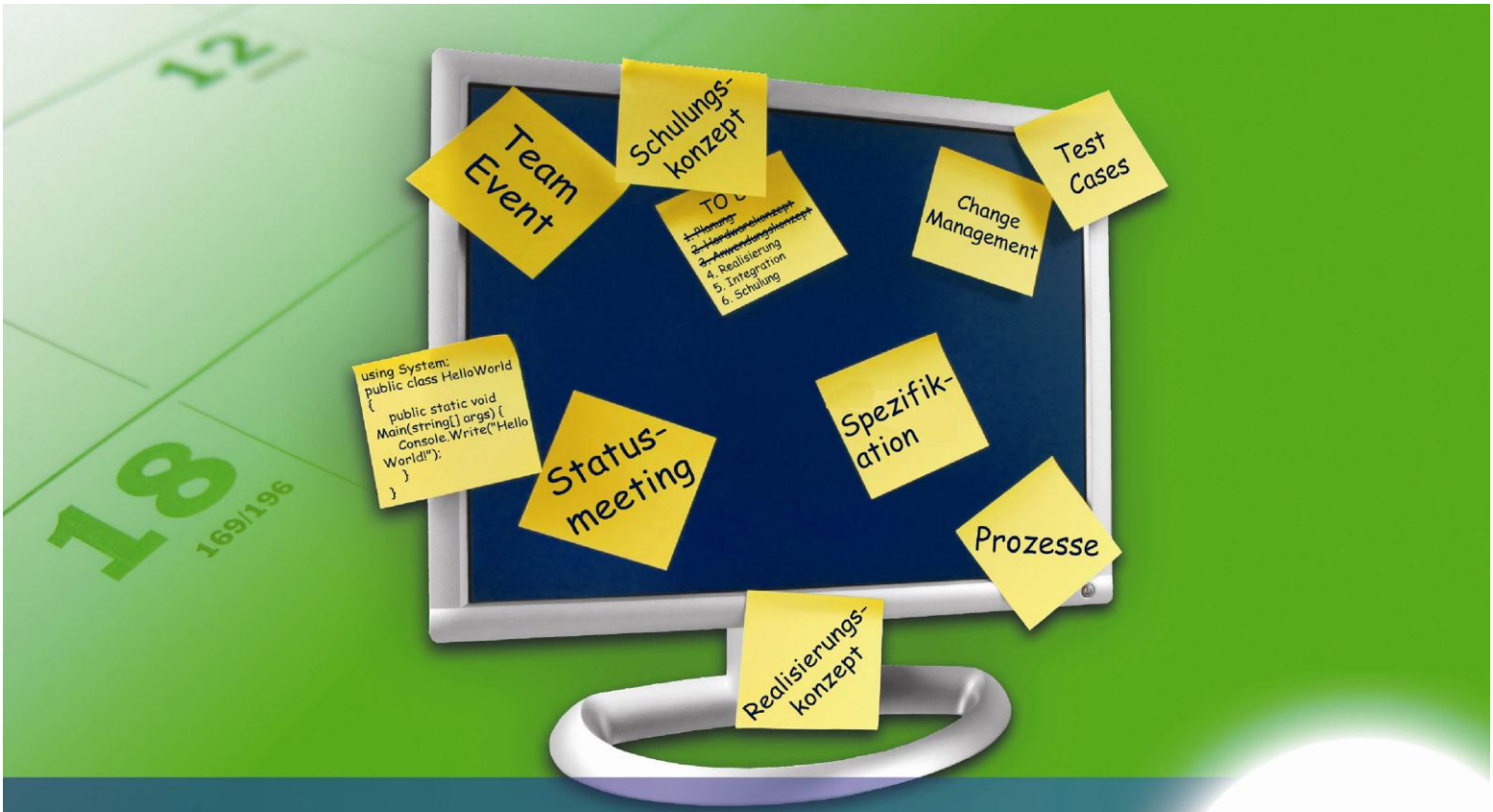

Mixin Show Case: Equals

Hands on Labs - Solution

rubicon
informationstechnologie gmbh

Table of Contents

1 Answers to 2.2.2 (Basic)

- **It is possible to implement Equals without overriding GetHashCode(). In which problems could you run into?**

  A hash code is a numeric value that is used to identify an object during equality testing. It can also serve as an index for an object in a collection.

  Without a GetHashCode implementation, the type cannot be added to HashTables.

- **There are five fields in class Address. Is there a generic way to compare each field per class?**

  As shown in the later sample, it is possible to use reflection to access all fields.

- **Does it make sense to compare the values of properties too to determine if two objects are equal?**

  Comparing properties can be troublesome. They could trigger some value changes. Consider lazy loading.

```
private Address _myAddress;
public Address MyAddress
{
  get
  {
    if (_myAddress == null)
    {
      myAddress = LazyLoader.LoadMyValue();
    }
    return myAddress;
  }
}
```

- **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)?**

  Consider the following sample: An importer retrieves data from csv files. Business rules require that timestamps are created on the import and stored with object. If two objects based on the same csv values are created at different times, are they non-equal just because the timestamp is different?

  Another approach would be to add algorithms to determine the likelihood that two entries are the same. For instance, you have an application, where more human users enter address data. They might enter for instance "Wall Str." and "Wall Street".

  Discussing this in full detail is out of scope.

- **We added an implementation of the interface IEquatable<Address> in Step 6. Please explain why this makes sense?**

  If you add this type to a List<T>, methods such as Contains, IndexOf, LastIndexOf, and Remove use an equality comparer for the list elements. The default equality comparer for type T is determined as follows. If type T implements the IEquatable<T> generic interface, then the equality comparer is the Equals(T) method of that interface; otherwise, the default equality comparer is Object.Equals(Object).

- **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?**

  True

- **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?**

  See the later exercise

- **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*?**

  Yes, but it is out of scope to discuss this in detail in this document.

2 Answers to 2.4.2 (Inheritance)

- **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"?**

  You cannot derive from another class. There is no multiple inheritance in .NET

- **Problem 2: Why does "StreetAddress.Equals()" return false? Hint: Which interface does StreetAddress implement?**

  StreetAddress implements IEquatable<Address>. Therefore only the fields of Address are compared.

- **We are using reflection in this sample. Do you think this can lead to performance issues?**

  Reflection is of course slower, but it is unlikely to run into serious performance problems.

3 Answers to 2.5.2 (Utility)

- **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 was calling this method).**

  What is your opinion about this statement? Have you encountered something similar?

  Subjective answer: Many developers, who have probably not too much experience with OOP, use static methods a generic cure for everything that cannot be implemented straight forward in OOP. Once the design and class hierarchies are suboptimal, this also seems to be the only cure.

- **Discuss the following enhancement**

  This can be seen as intermediary step to the strategy implementation

- **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?**

  One possibility would be to change the parameter list of the delegate method and add a reference parameter.

4 Answers to 3.1.2 (Mixin)

- **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>: Acts as a base class for mixins that require a reference to their target object (<see cref="Target"/>).

Target: Represents the target class.

OverrideTarget: Indicates that a mixin member overrides a virtual member of the mixin's target class.

- **Mixin<T> is required to reference the Target class. If no "Target is needed", is it possible to remove this base class?**

  Yes. The base class Mixin<T> just adds the reference to the Target.

- **The re-mix assemblies have references to other assemblies (such as Microsoft.Practices.ServiceLocation, Castle.Core, Castle.DynamicProxy2). Find out what these assemblies do!**

  Castle: See also http://www.castleproject.org/

  Microsoft.Practices.ServiceLocation.dll: This assembly contains the Common Service Locator interface used by the Composite Application Guidance to provide an abstraction over Inversion of Control containers and service locators; therefore, the user can change the container implementation with ease.

5 Answers to 3.1.4 (Mixin Attribute)

- **Use Reflector or re-motion Source Code to investigate `BindToTargetType!` What does this attribute do?**

  BindToTargetType indicates that a generic parameter of a mixin should be bound to the mixin's target type (unless the generic parameter type is explicitly.

6 Answers to 3.1.6 (Mixin Initialization)

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

  The factory methods cannot be dynamically created. So it is necessary to provide a huge number of possibilities.

7 Answers to 3.1.8 (Equal-Methods)

- **Can you explain what has changed in Task 4?**

  The first implementation implements the IEquatabley<T> implicitly. So there is a cast and the Equals method of the IEquatable is called. That's why it would not work, if do not cast to (IEquatable<T> in the bool Equals(object other) method

  In the second imeplementation, the IEquatable Interface is not called implicitly

- **Does it make sense to make public book Equals(T other) virtual?**

  This depends on your subclass.

8 Answers to 3.2

- **In which other uses cases it might make sense to use Mixins?**

  One example would be to implement IComparable.

Another possibility would be the following idea.

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

But not every Domain Object needs everything. There is no approval for changes to a person required, whereas an approval is obligatory to Documents. Does it make sense to replace inheritance with mixins, such as in the following example? (remark: The 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
```

It is out of scope to discuss this in detail, but there are many advantages with this scenario.

- **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?**

  A good starting point to research a possible use case would be a look on the DCI concept (http://en.wikipedia.org/wiki/Data,_Context_and_Interaction)