Generalized Implementation of Equals() for Mixed-Type Comparisons in C# .NET

Robert Dollinger
Computing and Information Systems
University of Wisconsin Stevens Point
Stevens Point, WI 54481
rdolling@uwsp.edu

Abstract

In order to be able to perform content based equality comparisons of Java/C# objects, programmers usually override the `equals()/Equals()` method defined in the `Object` class. Langer&Kreft [3] [4] showed that the ability of correctly comparing objects of different types requires that `equals()/Equals()` satisfies the reflexivity, symmetry and transitivity properties. They provide a sound and consistent equality check for Java class hierarchies, which requires each class to implement a recursive navigation method called, `_navigateClassHierarchy` that performs the non-trivial task of navigating up and down the inheritance tree in order to make sure that objects on different levels or even on different branches of the inheritance tree are compared correctly. In this paper, we show how, by using reflection and late-binding, one can develop a navigation method that is general, and totally independent, of any particular class. The implementation is provided in C#. NET, but a Java implementation is equally conceivable.

1 Introduction

C# programmers learn that whenever they define a new class, there are three things to be taken care of: override the `ToString()` method, override the `Equals()` method, and override the `GetHashCode()` method. All three methods are provided a default implementation at the very top level of the object hierarchy, in the definition of the Object class. In most cases, this is not what we need for our applications, hence the need to implement the overrides. Overriding `ToString()` is easy, and almost everybody seems to be comfortable with this task. Nobody seems to really know what to do with `GetHashCode()` and why it needs to be overridden. This is a long story and there are some good articles (most of them in the Java literature where the problems are very similar) dealing both with the reasons of implementing `GetHashCode()` and with various ways of doing it [2]. With `Equals()`, things seem to be simple and straightforward, but they are not. Most programmers tend to consider the implementation of `Equals()` a trivial task, overlooking many of the subtle issues that are involved. This results in incorrect implementations of the equality comparisons with hard to predict implications over the code using them. Errors caused by faulty implementations of `Equals()` may have profound implications and are difficult to trace and fix. Object equality comparison is such a basic operation, and it is used in so many places (e.g. collections management), that it is no surprise how much damage it can cause if not correctly implemented.

The problem with equality comparisons on objects has been identified for sometime and there are many articles dealing with it in various contexts and various ways [1][3][4][6]. These mostly end up providing recommendations of how to implement the equality test in some given limited contexts, rather then giving a comprehensive and "correct" solution. The ultimate solution, if there is one, is still not available. In our paper, we deal with the equality comparison problem in a more extended context, that is, under the assumption that objects of different classes in the same hierarchy can and should be compared for equality, providing meaningful and consistent results. Mixed type comparisons of objects have been dealt with by Langer&Kreft in [3] and [4], where they provide a correct Java solution to the problem. Their solution requires that each class joining the hierarchy of comparable objects implements two methods. The first one is called `_compareFields(Object other)`, and solves the specific task of comparing the fields of `this` object against the `other` object given as parameter. The second one is a recursive navigation method, which they called `_navigateClassHierarchy that` performs the non-trivial task of navigating up and down the inheritance tree in order to make sure that objects on different levels or even on different branches of the inheritance tree are compared correctly. Technical constraints, like using the current class name and invoking the parent class, forced the implementation of the navigation method in each class of the hierarchy to be done in a more or less specific way. However, from a conceptual point of view, the navigation method has nothing to do with any particular class. Furthermore, the burden of implementing the navigation method each time one wants to add a new class to the hierarchy makes this approach less appealing in practice.

We will show how, by using reflection and late-binding, one can develop a navigation method that is general and totally independent of any particular class. Thus, one can

factor out the navigation method from the classes in the hierarchy and implement it only in the root class of the hierarchy. All classes will inherit and use this method without any change. The implementation is provided in C#. NET, but a Java implementation is equally conceivable.

The remainder of this paper is organized as follows: in section 2, we introduce the *equals contract,* which specifies the correctness criteria for all implementations of equality comparisons, and deal with the pitfalls of those implementations that ignore this contract. In section 3, we bring arguments that mixed typed comparisons are unavoidable and we also give a proper semantics for such kind of comparisons. Section 4 provides the details of implementing mixed type comparisons in a class independent way. The key techniques for developing a general navigation method are reflection and late binding. Section 5 contains conclusions and some ideas of further developments.

2 The "equals" Problem

All .NET classes inherit the `Equals()` method from class `Object` in order to enable them to perform equality tests. In its default implementation, `Equals()` tests for object identity and is similar to the == operator. For this reason, it is common for new classes to override `Equals()` in order to provide meaningful and correct semantics to object comparisons, usually based on the values of data members in the compared objects. The minimal requirements users have to satisfy when overriding `Equals()` are specified in what is known both in the Java and .NET documentations as the *equals contract*.

2.1 The Equals Contract

The equals methods we are expected to implement are supposed to behave like any equivalence relation among the objects we compare. First of all this means that `Equals()` should be:

1) *Reflexive*: i.e. `x.Equals(x)` returns **true.**
2) *Symmetric*: i.e. `x.Equals(y)` returns the same value as `y.Equals(x)`.
3) *Transitive*: i.e. `(x.Equals(y) && y.Equals(z))` returns **true** if and only if `x.Equals(z)` returns **true**.

In addition to these, the *equals contract* specifies that successive calls of `x.Equals(y)` return the same value as long as the objects referenced by `x` and `y` are not modified, and `x.Equals(null)` always returns false.

The main point of the *equals contract* is that programmers are free to implement their concept of equality in any way they want, as long as the above rules are satisfied for the entire set of objects on which the equality test may be applied. Failure to comply with any of the rules produces incorrect implementations of `Equals`(), no matter how "correct" they may seem, and results in subtle bugs which are hard to fix.

2.2 Pitfalls in Implementing Equals()

In spite of most expectations and in spite of the apparent simplicity of the rules in the *equals contract*, providing correct implementations of `Equals()` is far from simple. This has caused quite a bit of disagreement in the programming community and, triggered a fair amount of debate.[1][3][4][6]

Everything is fine and looks simple as long as we deal with objects that belong to the same class. Take for example the sample code in Figure 1. The code is straightforward and one can check that it correctly compares objects of class A. For simplicity and sake of generality, we packaged the work of comparing the specific fields of A objects into the `CompareFields()` method.

```
public class A
{
	//field definitions
	bool CompareFields(Object other)
	{
		//code to compare specific fields
	}
	public override bool Equals(Object other)
	{
		if(other==this) return true;      //same object
		if(other==null) return false;     //nothing equals null
		if(!(other is A)) return false;   //incompatible types
		return CompareFields(other);      //compare the fields
	}
}
```

Figure 1: Implementation of Equals() in simple class A

The first problems with the above approach show up in the presence of inheritance. Let us assume that class B extends class A and implements its own override for `Equals()`. A very common way of doing this is illustrated by the code in Figure 2.

```
public class B:A
{
	//field definitions
	bool CompareFields(Object other)
	{
		//code to compare specific fields
	}
	public override bool Equals(Object other)
	{
		if(!(other is B)) return false;              //incompatible types
		if(!base.Equals(other)) return false;        //compare base class
		return CompareFields(other);                 //compare the fields
	}
}
```

Figure 2: Implementation of Equals() in inherited class B

The `Equals()` method in class B makes a call to the `Equals()` method from its base class:

```
	if(!base.Equals(other)) return false;
```

thus making sure that all its fields, including those inherited from class A, are properly tested. Testing for object identity and null reference are no longer needed here since these are going to be done by the base class method. After comparing its inherited fields, the method terminates by a call to its own `CompareFields()` method.

Although it may seem right, the above implementation of `Equals()` in class B is incorrect. Indeed, if `a` is an instance of class A and `b` is an instance of class B having all common fields set to similar values, then the expression `a.Equals(b)` evaluates to true, while `b.Equals(a)` evaluates to false. This means that the symmetry rule of the *equals contract* is violated. The reason is that, in the first case we are using the `Equals()` from class A, which succeeds because object b is an instance of its base class A. This is called a *slice comparison,* since only the slice of object b that is inherited from class A gets compared. Obviously, the second test will always fail because object `a` in not an instance of class A. In the first case, we compared objects a and b as class A instances and succeeded, while in the second case we attempted to compare them as class B instances, which is always supposed to fail.

In order to fix this problem, let us agree to compare objects as class A objects whenever one of them is an instance of class A. For this we only need to add an additional test as the first line of `Equals()` in class B:

```
if(other is A) return other.Equals(this);       //test after reversing operands
```

that is, if `other` is an instance of class A, then call `Equals()` for that class such that the objects are compared as instances of class A, thus ignoring fields defined in class B. With this adjustment, now both expressions `b.Equals(a)` and `a.Equals(b)` evaluate to true.

With this last modification our implementation of `Equals()` is symmetric, but is still incorrect. As we already mentioned, the *equals contract* is not easy to comply with and, as we are going to show, our implementation still violates the transitivity rule. To illustrate this let us take three objects `a`, `b1` and `b2` with the following properties:

- `a` is an instance of class A;
- `b1` and `b2` are instances of class B;
- `a`, `b1` and `b2` have the same values for their common fields (fields of class A);
- there is at least one field defined in class B for which `b1` and `b2` have different values.

Based on the above properties, it's easy to see that object `b1` and `b2` are not equal, and the expression `b1.Equals(b2)` will correctly evaluate to false. On the other hand, the expressions `b1.Equals(a)` and `a.Equals(b2)` both evaluate to true, and by transitivity we should have `b1.Equals(b2)` evaluate to true as if objects `b1` and `b2` would be equal, which contradicts our previous assertion.

The point of all this discussion is that after several fixes, we still do not have a correct implementation of `Equals()`. Fixing the transitivity problem is not easy, and it involves trade-offs on which there is still much disagreement in the programmers' community. In

fact, let us observe that even our solution for the symmetry problem is incomplete. It works, indeed, if one of the objects is the ancestor of the other one, but what if they are instances of classes located on different branches of the class hierarchy?

3 Mixed Type Comparisons

All the problems illustrated in section 2.1 would disappear if we eliminate the possibility of comparing objects of type A with objects of type B that is, if we forbid *mixed type comparisons*. By allowing equality comparisons only between objects that are precisely of the same type, both the symmetry and transitivity problems disappear, and it becomes relatively easy to provide correct implementations of `Equals()`. The changes in code are straightforward, since all we have to do is to replace the tests involving the `is` operator by a much stricter test using the `GetType()` method. For example, in class A the line:

```
if(!(other is A)) return false;   //incompatible types
```

would be replaced by:

```
if(other.GetType()!=GetType()) return false;   //if not the same type then fail
```

With this kind of modification in `Equals()`, attempts of comparing objects of different types will systematically fail, and one can check that this provides a sound and consistent behavior of `Equals()`, which, as such, is correct by the terms of the *equals contract*.

3.1 Mixed Type Comparisons Should be the Norm

Most of the articles about equality comparisons end up with the above mentioned easy way approach of disallowing mixed type comparisons in order to avoid the conceptual problems that come with the development of a more general, yet correct, implementation of `Equals()`. Mixed type comparisons would be permitted only in some particular cases, while alternative approaches are proposed for some other situations. All these end up in a mixture of partial or limited solutions, and/or sets of recommendations of when and how to implement equality tests in various circumstances. This is far from any useful and comprehensive solution of the problem at hand.

In [4] Langer&Kreft bring a first argument in favor of allowing mixed type comparisons, at least in some selected cases when the semantics of the application requires that. They still consider same type comparisons as the recommended way of implementing equality tests, and advocate for the use of mixed typed comparisons only when needed. The decision of which way to compare objects would be based on the semantics of the classes they instantiate. According to their examples, it makes sense to use mixed type comparison in a hierarchy where `Student` and `Employee` are subclasses of `Person`, since it would allow comparing a `student` to an `employee` to see if they are the same `person`. On the other hand, it would make little or no sense to compare an `apple` to a `pear` in a hierarchy in which `Apple` and `Pear` are subclasses of `Fruit`.

Clearly, one cannot ignore or totally avoid mixed type comparisons: this is far too limiting. On the other hand, using one or other type of comparability (same type or mixed type) based on circumstances like the semantics of the class or class hierarchy in an application, seems to be causing more problems than it is solving. On what basis would someone decide what kind of comparability to use for a given application or a given set of classes? Could this change over the development stages of the application? Could we use both kinds of comparability types in the same application? If, yes how would these interact or coexist?

Given these complications, why wouldn't we go beyond the hesitant position of using both same type and mixed type comparisons? If we adopt mixed typed comparisons as the norm, same type comparisons would be just a particular case of it, and the entire issue of object comparisons suddenly gets simplified. In fact, it becomes a purely technical problem of if and how it can be done right. The semantic argument is always debatable and not very productive in this case. After all, why would not we compare apples and pears, lets say in terms of the amount of vitamins brought into our body when consumed?

In the rest of our paper, we show that the idea of considering mixed type comparisons as the norm is realistic and can be technically supported in a way that is acceptable for programmers. Before this, we need to understand what it means to compare objects of different types, that is, we need to understand the semantics of mixed type comparisons.

3.2 Semantics of Mixed Type Comparisons

In section 2.2, we have seen that by comparing objects of different types, we ran into conflict with the *equals contract* by violating the symmetry and/or transitivity rules. Is there a meaningful way of doing such comparisons that would also comply with the *equals contract*? The answer is yes. We only need to carefully observe the rules of the *equals contract* in order to find out how.

First, let us observe that a necessary condition for two objects to be equal is to have all their common fields equal. Second, the transitivity rule is violated if we repeatedly compare objects of different classes by ignoring their subclass specific fields. In the example of section 2.2, we compared object `b1` with `a` and object `a` with `b2` by testing only the common fields in these objects, and ignoring the possible differences between `b1` and `b2` coming from fields defined in class B. This caused the violation of the transitivity rule.

The problem can be solved if we add the additional requirement that subclass objects have default values for all non-common fields. That is, in order to have object `b` equal with object `a,` they must have equal values for their common fields and object `b` must have default values for all other fields. These default values *are the same* for all objects of a given class. So, if `b1.Equals(a)` and `a.Equals(b2)` are `true` that means `b1` and `b2` have the same values not only for their fields which are common with object a, but

they also have the same default values for the rest of their fields, which means that `b1.Equals(b2)` is true, and thus transitivity of `Equals()` holds.

To summarize, the semantics of mixed type comparison can be formulated as follows:

Two objects compare equal if and only if:

1) *They have equal values for all their common fields;*
2) *Default values for all other fields.*

This semantics is consistent with the *equals contract* and uniformly applies on all objects of a class hierarchy. For example, given two objects on different branches of the class hierarchy, like student and employee, we would have equality if they have equal values for all their `Person` fields and default values for whatever fields are defined in the `Student` and `Employee` classes. As a special case, two objects with all their fields set to their default values will be equal even if they have no fields in common, that is they have totally different descriptions. This result may seem quite counterintuitive, but is perfectly consistent with the *equals contract*. Semantically, this may be interpreted like “if two objects contain no relevant information (i.e. all defaults) then they are the same”.

4 Implementing Mixed Type Comparisons

Based on the semantics defined above, Langer&Kreft[4] propose an approach that allows mixed type comparisons of objects in a well-defined class hierarchy. Their solution relies on two methods each class in the hierarchy is required to implement: a method for comparing fields, and a navigation method that makes sure that objects on different levels or even on different branches of the inheritance tree are compared correctly. They provide a Java implementation of the entire solution, which turns out to be compliant with the *equals contract*. Briefly, their approach can be characterized as follows:

- there is a class called `RootClass` on top of the hierarchy allowing mixed type comparisons, which contains a unique implementation of the equality comparison (`equals()` in Java). No other class implements the equals() method, all classes in the hierarchy inherit this method without any change.

- each class implements a method to compare its fields with the fields of an object given as parameter. This is a class specific method, since each class is responsible for correctly comparing the fields it defines.

- each class implements its own navigation method. This is a recursive method containing calls to the navigation methods of other classes in the hierarchy as well as calls to the fields comparing methods.

One can observe the trade-off Langer&Kreft propose in their approach. User classes are no longer required to implement the challenging equality comparison method. This is

implemented in the `RootClass`. Instead, classes joining the hierarchy are required to implement the fields comparing method and the navigation method.

The field comparing method is class specific, and its implementation in each class is a natural requirement.

The situation is different with the navigation method. Its implementation is not straightforward and can be almost as difficult as implementing the equality comparisons. Even if Langer&Kreft provide sample implementations of this method, understanding the mechanics of how everything works is quite a challenging task. This is hardly something that any user would like to deal with whenever defining a new class. Unfortunately, this makes the proposed solution quite unappealing, if not impossible to use in practice.

Given the difficulties to deal with the navigation method, we propose a new and more general solution which would be more practical from the users' point of view. The general layout of our approach follows the one proposed by Langer&Kreft, except that we provide a generalized and class independent implementation of the navigation method. We achieve this by using reflection and late binding techniques. Our implementation is in C#, but it can be easily translated to Java as well. The result is that the navigation method can be factored out to the top of the hierarchy, i.e. the `RootClass`. There will be only one implementation of this method, and it will be inherited by all classes. Consequently, the trade-off for the user classes is a much more practical one: a new class joining the hierarchy is required to implement only the `CompareFields()` method. Both `Equals()` and the navigation method are implemented in the `RootClass`.

4.1 Implementation of Equals()

For a given hierarchy of classes that allows mixed typed comparisons, there is one single implementation of the `Equals()` method, and that is located in the `RootClass`. Figure 3 shows the implementation of `Equals()` in the `RootClass`.

```
public class RootClass
{
   public override bool Equals(Object other)
   {
       if (other == this) return true;          //same object
       if (other == null) return false;         //nothing equals null
       if (!(other is Root)) return false;      //incompatible types
       return NavigateClassHierarchy(this.GetType(),(RootClass)other,true);
   }
}
```

Figure 3: Implementation of `Equals()` in the `RootClass`

The code is quite straightforward and class independent in spite of the fact that it may be called from each class in the hierarchy. After a couple of routine tests included for the sake of efficiency, the call is made to the navigation method, called `NavigateClassHierarchy()` which will be explained in section 4.3.

4.2 Implementation of the CompareFields() Method

The `CompareFields()` method is class specific and it is the only one that needs to be implemented in each class. This method is the materialization of the mixed type comparison semantics as presented in section 3.2 for the slice of fields defined in the current class. Basically, this method compares the fields defined in the current class with the corresponding fields of the object, called `other`, given as parameter. This is possible only when the currently defined fields are common with `other`, which means that `other` needs to be an instance of this class or of one of its subclasses. When this is not the case, one would check that the currently defined fields are set to their default values. A sample implementation of the `CompareFields()` method for a class called `MySubClass` extending class `MyClass` defining two fields, `field1` and `field2` is shown in Figure 4.

```
public class MySubClass:MyClass
{
   const int field1default=0;
   const int field2default=0;

   private int field1=field1default;
   private int field2=field2default;

   public new bool CompareFields(Object other)
   {
       if(other is MySubClass) //check fields if other is at least type of this
              {
                     if(field1!=((MySubClass)other).field1
                      ||field2!=((MySubClass)other).field2)
                            return false;
              }
              else   //check defaults otherwise
              {
                     if(field1!=field1default
                      ||field2!=field2default)
                            return false;
              }
              return true;
   }
. . .
}
```

Figure 4: Sample Implementation of the `CompareFields()` Method

Note how simple and straightforward the structure of this method is. All versions of `CompareFields()` would follow the above pattern; the only variations consist in the names and number of fields defined in the current class.

4.3 The Navigation Method

Since both `Equals()` and `CompareFields()` turn out to be fairly simple, one can expect that most of the functionality involved in the mixed type comparison operations is

concentrated in the `NavigateClassHierarchy()` method. As its name may suggest, this method will navigate the inheritance tree in order to check all fields of the compared objects. Indeed, all versions of the `CompareFields()` method perform strictly local operations that involve only the fields defined by the current class. In order to have the inherited fields compared as well, one will need to perform some kind of navigation across the inheritance tree in order to call the adequate version of the `CompareFields()` method. This is exactly the task of the `NavigateClassHierarchy()` method. In order to understand how this method works, let us take as an example the sample hierarchy depicted in Figure 5.

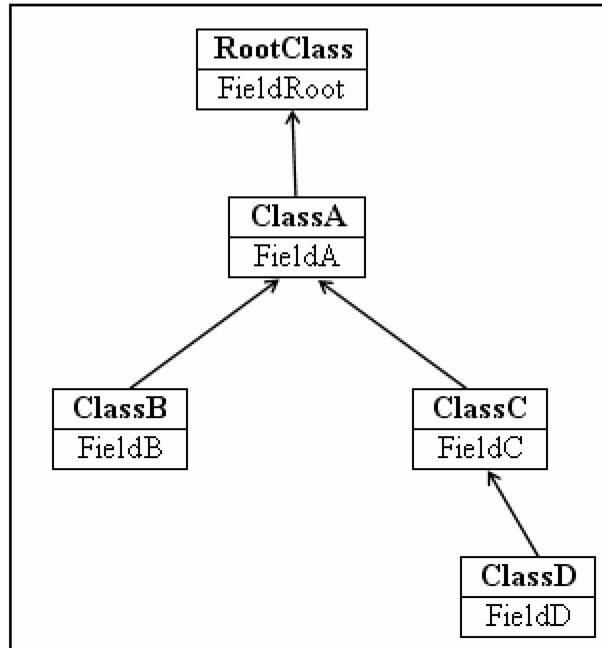

Figure 5: Sample Class Hierarchy

The sample class hierarchy is composed of the `RootClass` on top of the hierarchy and classes: `ClassA`, `ClassB`, `ClassC` and `ClassD` as depicted in the Figure. For simplicity, let us assume that each class defines exactly one field of its own: `FieldRoot` is defined by the `RootClass, ClassA` defines `FieldA` and so on. As a result, an object `b`, instance of `ClassB`, will have 3 fields: `FieldRoot` and `FieldA` as inherited from `RootClass` and ClassA and, `FieldB` defined in `ClassB`. Similarly, an object `d` instance of `ClassD`, will have fields: `FieldRoot`, `FieldA`, `FieldC` and `FieldD`.

Based on the semantics defined in section 3.2, two objects will be equal if they have equal values for their common fields and all their subclass specific fields are set to default values. For example, objects `b` and `d`, instances of `ClassB` and `ClassD`, are equal if and only if their `FieldRoot` and `FieldA` are equal (`b.FieldRoot==d.FieldRoot|| b.FieldA==d.FieldA`), and the specific fields: `FieldB` in `b` and, `FieldC` and `FieldD` in `d` are set to their default values.

Let us observe that the most general case of equality comparison is when the two objects are instances of classes located on different branches at different depths in the class

hierarchy. All other are particular cases of this one. So, keeping things simple, let us use the example in Figure 5 and analyze what is to be done when comparing objects `b` and `d`, instances of `ClassB` and `ClassD`. This means that one of the calls `b.Equals(d)` or `d.Equals(b)` is issued, and one can observe that in both cases, we end up with a call to the `NavigateClassHierarchy()` method. During the navigation across the class hierarchy, there are three main tasks that need to be solved:

- check for the default values of the subclass specific fields of object `b` – this requires navigation on the left side branch from `ClassB` to `ClassA`.
- check for the default values of the subclass specific fields of object `d` – this requires navigation on the right side branch from `ClassD` to `ClassA`.
- check for equality of the common fields - this requires navigation on the common branch from `ClassA` to `RootClass`.

Navigation will be done from the subclass levels to the upper level classes in the direction indicated by the arrows in Figure 5, by simply calling the `NavigateClassHierarchy()` method of the `base` class. At each step of the navigation process, the corresponding `CompareFields()` method is called in order to check the fields defined at the current level.

One of the challenges during the navigation process across the left or right side branch is to detect where each branch ends. This is the lowest level class that is also common for both branches, which in our example is `ClassA`. This class has the property that both objects `b` and d are instances of `ClassA`, i.e. `b is ClassA` and `d is ClassA` evaluate both to `true`. So, when navigating on the left side branch from `ClassB` to `ClassA`, one would stop when the `other` object (`d` in our example) is an instance of the current class. Unfortunately, using class specific tests of the form `b is ClassA` or `d is ClassA` makes the navigation method itself to be class specific, which means that a specialized version of the `NavigateClassHierarchy()` method would be required for each class. This is where *reflection* comes in providing the functionality needed to generalize the navigation method by avoiding direct references to the class names. First, one can use the `GetType()` method in order to get the type of a class or the class type of a given object. In our case the `Equals()` method (see Figure 3) will get the type of the current object, i.e. `this` and pass it as the first parameter of the `NavigateClassHierarchy()` method:

```
return NavigateClassHierarchy(this.GetType(),(RootClass)other,true);
```

while the navigation method is designed to receive as first parameter a `Type` object which will always be set to the type of the current class:

```
public bool NavigateClassHierarchy(Type typeOfThis,RootClass other,
                                                          bool reverseOrder)
```

The second useful functionality is that `Type` objects come with a method called `IsInstanceOfType()`, which is the counterpart of the `is` operator. This means that given the variable `typeOfThis` that holds the type of the current class, the expression

`typeOfThis.IsInstanceOfType(other)` will tell us when the `other` object is an instance of the current class that is, when we are at the end of the branch.

Another problem with the navigation method is to make sure that all the required branches are processed and that they are processed one single time. The technique proposed by Langer&Kreft [4] is to use a flag variable to control the navigation process. This is exactly what the third parameter of `NavigateClassHierarchy()` does. Initially, the navigation method is called with parameter `reverseOrder` set to `true`. When reaching the common class for the first time, the first two parameters are reversed and `reverseOrder` is set to `false`. In our example, this means that the current class type becomes `ClassB` while parameter `other` will be object `b`. This sets the right values for navigation along the right side branch. When returning from the right side branch, navigation simply continues on the common branch up to the `RootClass` level. If this level is reached, the entire comparison process successfully terminates and the two compared objects are equal.

Up to this point the `NavigateClassHierarchy()` method would look like this:

```
public bool NavigateClassHierarchy(Type typeOfThis, RootClass other, bool
                                                                     reverseOrder)
{
   if (typeOfThis.IsInstanceOfType(other)&& !reverseOrder)
   //reverse order
      return NavigateClassHierarchy(other.GetType(),this,true);
   //compare my fields
   ...        code for comparing fields to be added here!
   //succesfully done when at RootClass
   if(typeOfThis==Type.GetType("EqualsImplementation.RootClass")) return true;
   //navigate to upper level
   return NavigateClassHierarchy(typeOfThis.BaseType,other,reverseOrder);
}
```

The last challenge in the development of a class independent navigation method is to compare the fields defined at the current class level, by issuing a call to the right version of the `CompareFields()` method. One may expect that some kind of cast operation by the current level class type may provide the binding to the right method. Unfortunately, this is not the case, since cast operations are meant to alter compile time binding. What we need here is to dynamically bind our call to the version of the `CompareFields()` method that corresponds to the current class level. That is, at each navigation step a different version of the `CompareFields()` method needs to be called. This kind of functionality can be achieved by using *late binding* techniques.

In C# there are two ways one can use to dynamically invoke a method: by using the `Invoke()` method, or by using the more general `InvokeMember()` method.

When using the `Invoke()` method, a required preliminary step is to get the method information corresponding to the method we want to call dynamically, that is `CompareFields()`. This kind of information is contained in a `MethodInfo` object which is returned by the `GetMethod()` method of a type object. In our case, the expression `typeOfThis.GetMethod("CompareFields")` returns a `MethodInfo` object describing

the `CompareFields()` method for the current level class type represented by the variable `typeOfThis`. The `MethodInfo` object is the one that provides access to the `Invoke()` method through which the right version of `CompareFields()` is called. The complete code for the dynamic call is as follows:

```
MethodInfo compareFieldsMethod=typeOfThis.GetMethod("CompareFields");
if(!(bool)compareFieldsMethod.Invoke(this,new Object[] {other})) return false;
```

The first parameter of `Invoke()` is the object making the dynamic call, while the second one is an array of objects representing the list of parameters with which the dynamic method is called.

The equivalent dynamic call sequence using `InvokeMember()` is:

```
if(!(bool)typeOfThis.InvokeMember("CompareFields",  BindingFlags.InvokeMethod  |
       BindingFlags.Default,null,this,new Object[] {other})) return false;
```

Details about `InvokeMember()` and its parameters can be found in [5] and [7].

The complete version of the `NavigateClassHierarchy()` method using `InvokeMember()` is given in Figure 6.

```
public bool NavigateClassHierarchy(Type typeOfThis, RootClass other,
                                                         bool reverseOrder)
{
   if (typeOfThis.IsInstanceOfType(other)&& !reverseOrder)
   // reverse order
   return NavigateClassHierarchy(other.GetType(),this,true);
   // compare my fields
   if(!(bool)typeOfThis.InvokeMember("CompareFields",BindingFlags.InvokeMethod
       | BindingFlags.Default,null,this,new Object[] {other})) return false;
   //succesfully done when at RootClass
   if(typeOfThis==Type.GetType("EqualsImplementation.RootClass")) return true;
   //navigate to upper level
   return NavigateClassHierarchy(typeOfThis.BaseType,other,reverseOrder);
}
```

Figure 6: Complete version of the `NavigateClassHierarchy()` Method

5 Conclusions and Further Developments

`Equals()` is intended to capture the semantics of content based equality comparisons of objects as opposed to object identity implemented by the == operator. This would make `Equals()` a class specific method, and such overriding it is expected to be an every day routine. However, correctly implementing `Equals()` turns out to be a challenging task, and everyday routines are not supposed to be challenging. The correctness criteria for equality comparisons are given by the *equals contract*. On the other hand, programmers would expect a solution that is both simple and free of artificial limitations. Programmers should not find a difficult challenge in implementing such a basic functionality like object equality and should not be limited to compare only objects of the same type. We provide an approach to the equality comparison problem which is a generalization of the

solution provided by Langer&Kreft in [4] and is able to reconcile the requirements of the *equals contract* with the legitimate expectation of programmers. This means that mixed type comparisons of object are allowed without limitations, while programmers are expected to implement a fairly straightforward `CompareFields()` method instead of having to override `Equals()`. The key techniques used are based on reflection and late binding, which allow class independent navigation of the inheritance tree.

There are several directions for further development of the work presented in this paper. First, based on the description in section 4.3, one can easily implement a more efficient non-recursive version of the navigation method. Space limitations forbid us from including it here. Second, the functionality of mixed type comparisons could be made generally available in a user transparent manner. All it would take is to implement both `Equals()` and `NavigateClassHierarchy()` methods at the level of the `Object` class. Providing these as standard system level functionality would leave programmers only with the requirement of implementing the `CompareFields()` method as their own local concept of object equality. This requirement could be enforced by having classes to implement an adequate interface defining the `CompareFields()` method. Given its simple structure, an even more convenient approach could be to automatically generate the code of the `CompareFields()` method based on a list of user designated fields, considered as relevant for the equality tests, along with their default values.

As a final thought, we would like to emphasize the idea that it may be beneficial to have more elaborate content-based default functionality implemented at system level both for `Equals()` and `GetHashCode()`. The two are closely related and a navigation technique similar to the one presented here could be used in the computation of objects' hash codes. As with field comparisons, users will only have to designate which fields would be used in the generation of the hash code value.

References

[1] Mark Davis (2000) – "Durable Java: Liberte, Egalite, Fraternite", *Java Report,* January 2000, URL: http://www.macchiato.com/columns/Durable5.html

[2] Mark Davis (2000) – "Durable Java: Hashing and Cloning", *Java Report,* April 2000, URL: http://www.macchiato.com/columns/Durable6.html

[3] Angelika Langer, Klaus Kreft (2002) – "Secrets of equals() – Part 1, Not all implementations of equals() are equal", *JavaSolutions,* April 2002, http://www.langer.camelot.de/Articles/JavaSolutions/SecretsOfEquals/Equals-1.html

[4] Angelika Langer, Klaus Kreft (2002) – "Secrets of equals() – Part 2, How to implement a correct slice comparison in Java", *JavaSolutions,* August 2002, http://www.langer.camelot.de/Articles/JavaSolutions/SecretsOfEquals/Equals-2.html

[5] Jesse Liberty (2003) – "Programming C#, Third Edition", *O'Reilly*, 2003.

[6] Andreas Schaefer (2004) – "All equals() are not born equal", *Java.net*, October 2004, http://weblogs.java.net/blog/schaefa/archive/2004/10/all_equals_are.html

[7] Andrew Troelsen (2003) – "C# and the .NET Platform, Second Edition", *Apress*, 2003.